POWER-AWARE RESILIENCE FOR EXASCALE COMPUTING

by

Bryan Mills

B.S. in Computer Science and Mathematics, Bethany College 2001

M.S. in Computer Science, University of Pittsburgh 2007

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2014

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Bryan Mills

It was defended on

May 30, 2014

and approved by

Taieb Znati, Ph. D., Professor

Rami Melhem, Ph. D., Professor

Daniel Mosse, Ph. D., Professor

Alex Jones, Ph. D.

Kurt Ferriera, Ph. D.

Dissertation Director: Taieb Znati, Ph. D., Professor

POWER-AWARE RESILIENCE FOR EXASCALE COMPUTING

Bryan Mills, PhD

University of Pittsburgh, 2014

To enable future scientific breakthroughs and discoveries, the next generation of scientific applications will require exascale computing performance to support the execution of predictive models and analysis of massive quantities of data, with significantly higher resolution and fidelity than what is possible within existing computing infrastructure. Delivering exascale performance will require massive parallelism, which could result in a computing system with over a million sockets, each supporting many cores. Resulting in a system with millions of components, including memory modules, communication networks, and storage devices. This increase in number of components significantly increases the propensity of exascale computing systems to faults, while driving power consumption and operating costs to unforeseen heights. To achieve exascale performance two challenges must be addressed: resilience to failures and adherence to power budget constraints. These two objectives conflict insofar as performance is concerned, as achieving high performance may push system components past their thermal limit and increase the likelihood of failure. With current systems, the dominant resilience technique is checkpoint/restart. It is believed, however, that this technique alone will not scale to the level necessary to support future systems. Therefore, alternative methods have been suggested to augment checkpoint/restart – for example process replication.

In this thesis, we present a new fault tolerance model called *shadow replication* that addresses resilience and power simultaneously. *Shadow replication* associates a shadow process with each main process, similar to traditional replication, however, the shadow process executes at a reduced speed. *Shadow replication* reduces energy consumption and produces

solutions faster than checkpoint/restart and other replication methods in limited power environments. *Shadow replication* reduces energy consumption up to 25% depending upon the application type, system parameters, and failure rates. The major contribution of this thesis is the development of *shadow replication*, a power-aware fault tolerant computational model. The second contribution is an execution model applying shadow replication to future high performance exascale-class systems. Next, is a framework to analyze and simulate the power and energy consumption of fault tolerance methods in high performance computing systems. Lastly, to prove the viability of *shadow replication* an implementation is presented for the Message Passing Interface.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

The race to build the world's first exascale-class system has been underway for the last ten years and many challenges remain. Two of the biggest challenges facing these future systems are resilience failures and the need to operate within power constraints, both a direct result of the massive amount of parallelism necessary to meet the requirements of exascale. Delivering exascale performance is projected to require a system with a million sockets, each supporting many cores [2]. Furthermore, achieving the expected level of performance will require many millions of components, including increases in memory modules, communication networks, and storage devices. With this explosive growth in the number of components will come an increase in the number of faults; reducing the overall system reliability and increasing the system's power requirements. In this chapter, we first discuss the functional and physical attributes of an exascale computing infrastructure, including a functional comparison with existing petascale systems. We then explore the challenges that need to be addressed to enable exascale computing. The last section of this chapter focuses on the thesis objective and contributions of this research.

1.1 EXASCALE COMPUTING SYSTEMS

We start by defining the concept of exascale and providing some background on the scalability of high performance computing (HPC). An exascale system, as defined by the Exascale Working Group [8], is a system that contains at least one system attribute that is one thousand times larger than that of today's petascale system. Typically, high performance computers are ranked by the number of floating point operations per second (FLOPS), which

is a functional performance metric. There is a direct relationship between this functional performance metric and the application performance, although the radical increase in the level of parallelism might prove to be challenging for today's applications – potentially limiting the gains achieved in functional performance. For example, increases in application observed failure rates could offset the improvement in functional performance. Most work in exascale computing research, including this thesis, focuses upon increasing functional performance. Specifically, this thesis is focused on providing resilience to the software and runtime environments that will enable applications to harness the available resources in terms of hardware, power, energy, and time.

In order to deliver the desired functional performance and effectively harness its capabilities, several daunting scalability challenges must be addressed. In the late 90's, terascale performance was achieved with fewer than 10,000 heavyweight single-core processors. A decade later, petascale performance required about ten times as many processors as terascale performance. Delivering exascale computing will require one million processors, each supporting 1,000 cores, resulting in a billion-core computing infrastructure while also requiring a dramatic increase in the number of memory modules, communications devices and storage components. Table 1 is the U. S. Department of Energy's projected specifications for future exascale-class systems, defining "swim lanes" representing expected paths to reaching exascale performance [2]. Note that there are two distinct paths toward achieving exascale, one requiring a million nodes supporting 1,000 cores, and the other having 100,000 nodes supporting 10,000 cores. In either case, exascale performance is achieved through unprecedented levels of parallelism, supporting systems with billions of computing components.

As today's HPC systems grow to meet the requirements of tomorrow's exascale-class systems, two of the biggest challenges are power consumption and system resilience. Regardless of the exact constraints under which the computing infrastructure must operate, power is undoubtedly a limiting factor in achieving the expected exascale performance required by the supporting applications. Addressing the prescribed power constraints is critical to the design and operations of exascale computing systems. The U. S. Department of Energy has already targeted a power limit of 20 megawatt [2], as indicated in Table 1. As a direct consequence of the increase in the number of components, the overall system resilience to

System Parameter	Petascale	Exascale (projection) Swim Lane 1	Swim Lane 2	Factor Change
System Peak	2Pf/s	1 Ef/s		500
Power	6MW	$\leq$ 20MW		3
System Memory	0.3 PB	32 - 64 PB		100-200
Total Concurrency	225K	$1B \times 10$	$1B \times 100$	40,000-400,000
Node Performance	125GF	1TF	10TF	8-80
Node Concurrency	12	1,000	10,000	83-830
Network BW	1.5 GB/s	100 GB/s	1,000 GB/s	66-660
System Size (nodes)	18,700	1,000,000	100,000	50-500
I/O Capacity	15 PB	32 - 64 PB		20-67
I/O BW	0.2 TB/s	20-60 TB/s		10-30

Table 1: Comparison of a petascale supercomputer to an expected exascale-class supercomputer, as defined by the U. S. Department of Energy Exascale Workshop [2].

faults decreases substantially. Regardless of the reliability of individual components, system resilience will continue to decrease as the number of components increases[1]. Achieving high resilience to failures under strict power constraints is a daunting and critical challenge that must be addressed to enable exascale systems. In the following, we further explore the dimensions of these challenges and their impact on the design and performance of future exascale-class systems.

1.1.1 Power Challenge

With the explosive growth in the number of components will come a dramatic increase in system power requirements. Figure 1 shows a steady rise in system power consumption to 1-3MW in 2008 but then a sharp increase to 10-20MW in the following years. The trend shows that system power consumption could surpass 50MW by 2016. This makes system power a leading design constraint on the path to exascale. The Department of Energy has recognized this trend and established a power limit of 20 megawatt [2], challenging the research community to provide a 1000x improvement in performance with only a 10x increase

[1]For example, a system consisting of 1 million components, each averaging a fault every 25 years, produces a system that will experience a fault every 10 minutes on average.

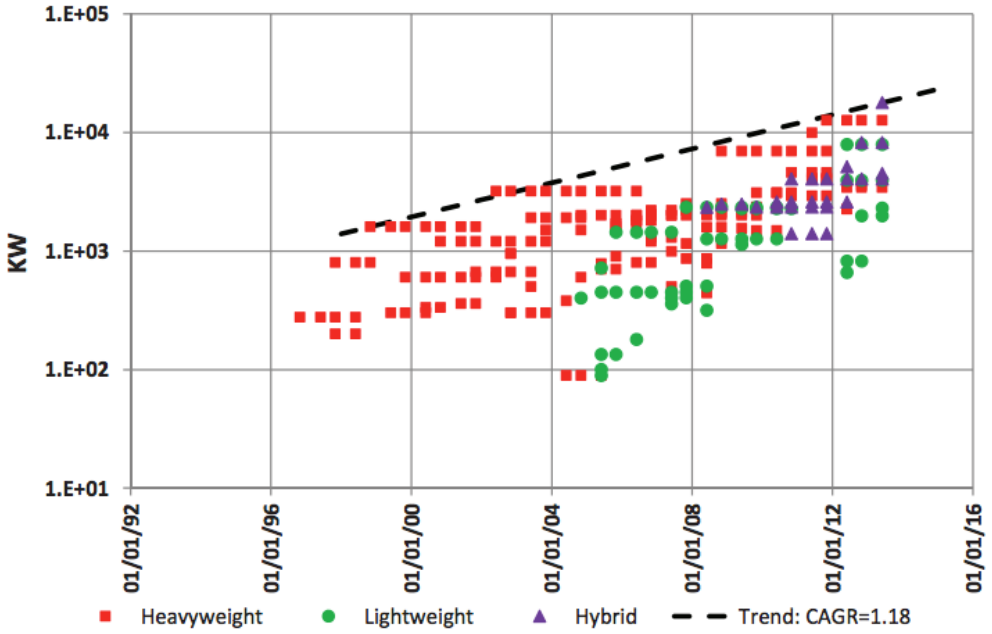


Figure 1: System Power Projections [57]

in power. This constraint is largely pragmatic, derived from an estimated cost of $1 million dollars per megawatt per year to operate a supercomputer, resulting in a power budget of $20 million per year for any one supercomputer. Regardless of the actual budget, it is clear that exascale systems will need to operate under constraints of a specific budget, which very likely will be more constraining than what current systems can achieve if they were to support the expected requirements of future exascale applications.

Adherence to power budget constraints presents a challenging dilemma in the operation of exascale-class systems. Exascale-class systems will be capable of achieving levels of power consumption that exceed their prescribed power budget constraints. For example the total power consumption of a system designed with 150,000 sockets, each consuming 200 watts of power at full speed, would consume 30 megawatts if all sockets were operating at full speed. To remain under the 20 megawatt limit, either 50,000 of these sockets must be powered off, or the power consumption of some or all of the sockets must be reduced. While this may seem inefficient, as more hardware is available than can be supported by the power

infrastructure, this will become a reality in exascale-class systems.

Power is the rate at which energy is being consumed, which for the purposes of this thesis is equivalent to the speed of execution [2]. The energy consumed by an application is a function of the time-to-solution and the amount of power consumed. Reducing the execution speed can reduce the power consumption, however, reducing the execution speed will also increase in the time-to-solution. The tradeoff between power consumption and time-to-solution is at the crux of energy research in computing systems. Although the intricacies of this tradeoff are complex, it has been demonstrated that energy savings is possible by reducing execution speeds [113, 53, 6].

The power-limits placed upon exascale-class systems is at the heart of the tradeoff between power consumption and time-to-solution. The limit is both pragmatic and practical, on one hand, it limits the energy costs and on the other hand it is reflective of the practical constraints of the infrastructure supporting exascale-class systems. In either case power-constraints are a reality that future exascale environments have to face, making the relationship between power and energy consumption increasingly more critical. This thesis presents a power-aware replication model, that has the potential to save energy, however, the tradeoff between power consumption and time-to-solution will remain an issue to be addressed.

1.1.2 Resilience Challenge

The field of fault tolerance in computing systems is well established, and significant advances on how to deal with faults have been achieved by different communities. State of the art solutions largely rely upon restarting the execution of the application. To avoid the full re-execution of the failing application, fault-tolerant techniques typically checkpoint the execution periodically; upon the occurrence of a fault, recovery is achieved by restarting the computation from a safe checkpoint [55].

However, recent work in our lab [76, 75] and others [38, 92, 10, 13] have shown that existing solutions are likely not to scale to the level of faults anticipated in exascale environments. Given the increase in system failure rates and the time required to checkpoint

[2]The relationship between execution speed and power consumption is well established and discussed in more detail in Chapter 2.

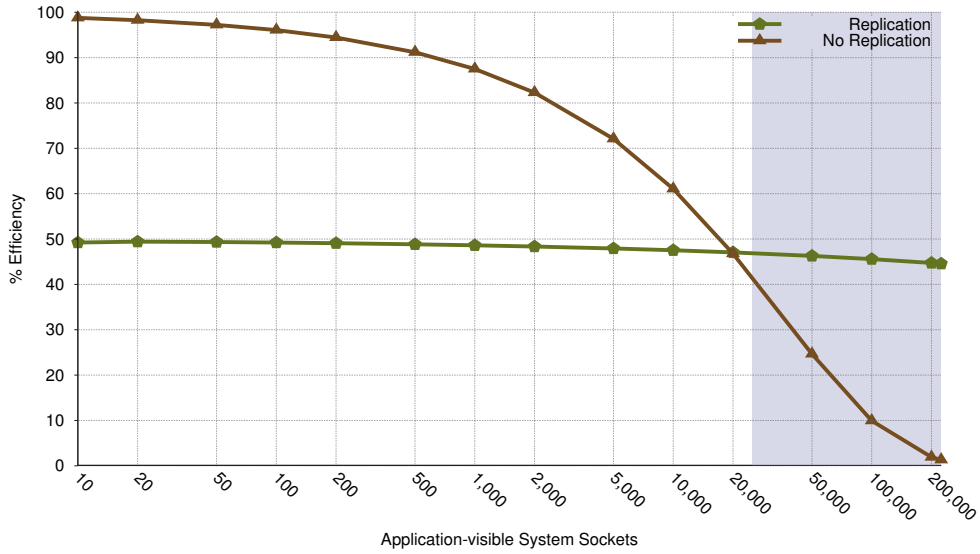


Figure 2: Effect on system efficiency as number of nodes increase. Node MTBF 5 years and a fixed checkpoint time of 20 minutes. Twenty minutes is an estimate of checkpoint time given the amount of system memory (32-64PB) and the I/O bandwidth (20-60TB) projected in the "swim-lanes" in Table 1, producing a checkpoint write time of between 9-54 minutes. This data was produced using a simulator [93].

large-scale compute- and data-intensive applications, it is very likely that the time required to periodically checkpoint an application and restart it upon failure may exceed the system mean time between failures [92]. Consequently, applications may achieve very little computing progress, thereby reducing considerably the overall performance of the system. Figure 2 illustrates this fact and shows that, using coordinated checkpointing, the system efficiency drops below 50% as the number of sockets increases. Several studies have shown this same behavior and proposed traditional replication as a possible solution [81, 92].

Replication, also referred to as state-machine replication, is a well-known technique that has been shown to scale to meet the resilience requirements of large distributed and mission-critical systems. Based on this technique, a processes state and computation are replicated across independent computing nodes. When the main process fails, one of the replicas takes over the computation task. Replication requires doubling the number of nodes, since each process must have at least 1 replica, thereby reducing the system efficiency to 50%. A major shortcoming of traditional replication in HPC is the increased power consumption caused by the need for additional resources to tolerate failure, which might exceed the power budget imposed upon exascale-class machines.

1.2 THESIS STATEMENT

Two of the biggest challenges facing exascale systems are power and resilience, both a result of the increase in the number of components. In this thesis, we will show that it is possible to develop a solution that addresses both of these challenges. Our goal is to justify and provide evidence to prove the following thesis statement:

"In a power-limited environment, it is possible to build a fault tolerant system for strongly scalable applications that is more time and energy efficient than existing fault tolerance techniques for exascale-class, high performance, computing systems."

We seek to achieve this objective by developing a new power-aware replication tech-

nique called *shadow replication*. Through the use of analytical models, simulations, and experimentations, we demonstrate that shadow replication provides system resilience more efficiently than both checkpointing and traditional replication in power-limited environments. To achieve a fair comparison with existing fault tolerance methods, we study the power and energy consumption of existing methods and suggest power saving techniques that could be applied in today's systems.

The basic idea of *shadow replication* is to associate with each process a "shadow process," whose execution speed depends on the performance requirements of the underlying application. A shadow process is an exact replica of the original process. In order to overcome failure, the shadow is scheduled to execute concurrently with the main process, but at a different computing node. In order to minimize energy, the shadow process initially executes at a decreased processor speed. The successful completion of the main process results in the immediate termination of all shadow processes. If the main process fails, the primary shadow process immediately takes over the role of the main process and resumes computation, possibly at an increased speed, in order to complete the task. The main challenge in realizing the potential of the shadow replication stems from the need to compute the speed of execution of the main process and the speed of execution of its associated shadows, both before and after a failure occurs, so that the applications requirements are met, while minimizing energy consumption.

Since the failure of an individual component is much lower than the overall system failure, it is very likely that most of the time the main processes complete their execution successfully. Successful completion of a main process automatically results in the immediate halting of its associated shadow processes, providing a significant savings in energy consumption in comparison with replication. The completion of a main or its shadow results in the successful execution of the underlying task.

1.2.1 Thesis Contributions

The thesis has five main contributions. First, a new power-aware fault tolerant computation model, referred to as *shadow replication*, is proposed and a formal definition of it execution

model in high performance computing is provided. Second, an analytical model is developed to compute the expected energy consumption of shadow replication, and an optimization framework is used to determine the energy-optimal execution speeds of the main process and its associated shadow, before and after failure occurs. Third, the analysis of the expected energy consumption of *shadow replication* and a comparison to other fault tolerant methods is presented. Fourth, a simulator is developed to validate the analytical models and provide further analysis. Fifth, to prove the viability of *shadow replication* within the Message Passing Interface (MPI).

The remainder of this section will highlight each of these contributions and discuss the main outcome related to their research.

A power-aware fault tolerant computational model called *shadow replication*. After exploration of the power and energy requirements of existing fault tolerant methods, it became clear that today's techniques will be inefficient in exascale systems. This study led us to develop the power-aware fault tolerant computational model of *shadow replication* and is the central focus of this thesis. To explore this technique we first develop a execution model and provide a sketch of how such a model could be deployed in the high performance computing environment. We then define our design space using this execution model and further refine the search space in the context of HPC.

Optimization framework for determining the energy, power, and time optimized execution speeds of *shadow replication*. Having defined the execution model, we develop an analytical framework that describes both the time-to-solution and energy requirements. Using this framework an optimization problem is constructed to derive the energy optimal execution speeds for *shadow replication*. To enable analytical comparison we also develop analytical energy and power models for coordinated checkpointing and traditional replication.

Comparison of the power and energy consumption of resilience methods. Armed with these models, we explore the potential energy savings achievable using *shadow replication*. Ultimately, we demonstrate that in exascale-class systems replication is significantly more efficient than coordinated checkpointing. Further, we show that *shadow replication* is up to 25% more efficient than traditional replication depending upon the system

parameters, the application's communication dependencies and the available laxity.

Additionally, using the analytical model we study the effect *shadow replication* has upon strong scaling applications in power limited environments, such as those expected in exascale systems. In a power limited environment, it is expected that there will be more available sockets than power to turn them on. Our analytical models show us that in this environment *shadow replication* can provide up to a 46% reduction in both time-to-solution and energy consumption over traditional replication. The main reason for this savings is that *shadow replication* enables the system to utilize more sockets than traditional replication and do so more efficiently than coordinated checkpointing.

A simulation framework for studying power and energy consumption of fault tolerance techniques. Given these positive analytical results, we implemented a simulator to validate our analytical model and to confirm the energy savings achievable by *shadow replication*. Simulation confirmed our 14-46% savings for applications with no communication, but showed a 2-24% savings in tightly coupled applications. After further investigation, we found that in exascale-class machines, our simulations were experiencing cascading process delays which drastically cut the potential gains we saw in our analytical models.

An implementation of *shadow replication* in the MPI library. To confirm the feasibility of executing *shadow replication* in exascale-class systems we provide an implementation in the HPC messaging passing interface (MPI). This implementation allows a non-modified MPI application to execute in HPC systems while taking advantage of our new fault tolerance method. In this thesis, we provide implementation details and a discussion of those issues related to the pairing of shadow replication and MPI. We then show experimental results using our implementation in small clusters that have the capability to measure power and energy usage.

A related contribution is presented in the Appendix addressing energy concerns faced in current systems. In this section, we study the power profiles of coordinated checkpointing in today's systems. Showing that there is a potential to save energy by reducing the CPU during the writing and restoring operation with little effect on time to solution of existing applications. This technique alone could save 5-10% of the energy consumption in today's systems.

1.3 THESIS ORGANIZATION

In Chapter 2 we provide a background study of the related work for fault tolerance and power management in high performance computing. Chapter 3 develops *shadow replication* as a new computation model and describes how such a technique would be implemented in high performance computing environments. Further, we develop an analytical framework and compare the power and energy consumption of *shadow replication* to other fault tolerant techniques in Chapter 4. Chapter 5 details a simulation framework developed to study power and energy consumption of resilience techniques and demonstrates potential energy savings of *shadow replication*. In Chapter 6 we provide the details of the MPI implementation. Finally, in Chapter 7 we provide some concluding remarks and detail future work and continued research being actively pursued in this area. The appendix presents experimental results and analysis of coordinated checkpointing and propose power-aware enhancements.

2.0 BACKGROUND AND RELATED WORK

This chapter provides background on fault tolerance and power management as it pertains to high performance computing (HPC), with a focus on exascale-class systems. First, a conceptual framework for how faults interact within the context of the system and set the stage for how fault tolerance is provided. It will also provide background on checkpointing and replication, with a focus on the most current research on these topics and how they apply to exascale-class systems. A review of power management concerns in HPC, are also discussed, along with current research in this area. As discussed in the previous chapter, *shadow replication* aims to simultaneously addresses fault tolerance and power management in HPC; we therefore review previous work in the interplay between fault tolerance and energy consumption. Lastly, within this context, we argue that *shadow replication* is positioned provide power-aware resilience by enabling tradeoff between hardware and time redundancy.

2.1 FAULT TOLERANCE

Before discussing how systems provide fault tolerance, it is necessary to define fault, failure, and error [52]. A *fault* is a deviation from the expected behavior of a system. There are two distinct types of faults: operational and design [52]. Operational faults are those that occur at the lowest level of the system and are usually the result of a physical malfunction, such as a memory cell being stuck at one. Design faults are those that occur as a result of the design being flawed, for example, a software bug. Typically, fault tolerance in HPC systems is built to address operational faults, but not design faults. This stems from the fact that operational errors can be identified and corrected at the system software level, whereas

design faults need to be addressed at the application or middleware level. Techniques such as design diversity can address operational errors; but these techniques are usually deployed at the application level [58].

A fault can be reproducible, meaning that every time a series of steps are followed the fault occurs. A fault could also be transient, meaning that it sometimes occurs but there is no specific set of steps that always produces the error. Transient faults can occur because of physical factors - for example, the heat of processor might cause it to perform the wrong operation. Faults can also be classified as hard or soft faults. Hard faults are those that result in a system becoming completely unusable and are therefore trivially reproducible; for example, a broken power supply would result in a hard fault. Soft faults are those which produce errors without completly halting operation. For example, a slowly running machine that can be corrected by rebooting.

Failures are faults that become visible to the application or the application user. There are cases where a fault is always visible to the application, causing the terms fault and failure to be used interchangeably. For example, a faulting power supply might always cause a computer to turn off which is always visible to the user of that computer, causing them to associate the fault with the failure.

An *error* is the system state that occurs after a failure. Typically one cannot directly observe a fault or failure but instead observes the resulting error state. There is a wide range of error states that might occur after a failure, for example an incorrect calculation might be returned or a segmentation fault could occur due to the application accessing an invalid memory location.

To summarize, faults are the flawed system component, failures are visible effects arising from faults, and errors are the system state after a failure. For example, were a processor to add two numbers, but instead subtracted those numbers, the processor would cause an operational fault. A failure occurs if the fault results in unexpected behavior as observed by the application. If, for example, the value of the numbers that were to be added were both zero the processor's fault remains undetected, and exhibits no deviation from the expected behavior. Therefore, this fault would not be considered a failure. An error is the state of the system that results after a failure occurs. In this example, the resulting state would

not be an error state, because the final result was not incorrect. If however, the numbers being considered were non-zero then a failure would have occurred, due to the incorrect summation.

Because the reasons for and manifestations of faults are extremely varied, faults are described using a fault model. There are a variety of fault models proposed in the literature. For our discussion, we will focus on three of the most popular fault models. A fail-stop model describes a faults in which a processor completely stops working upon failure and is easily detected by its neighbors. A crash model exhibits the same behavior as a fail-stop model, but assumes that it might be harder for its neighbors to recognize the fault. A Byzantine fault model is one in which a system may continue to execute, but produce random or malicious outcomes, resulting in unexpected behavior. In this short list of models, Byzantine is the most generic because it covers the behavior of the other two models, whereas fail-stop is the most restrictive model because failures can be easily detected.

2.1.1 Reliability Metrics

There are several metrics commonly used when discussing the resiliency of HPC systems. The reliability of a component describes the probability that the component will perform its intended function during a specified period of time. The failure rate, λ, is the frequency with which a component will experience failures.

The other commonly used metric is the Mean Time Between Failure (MTBF), which describes the mean time between any two consecutive failures. The MTBF is the inverse of the failure rate, λ. For a typical hardware component this is defined in a number of years. This time is equal to the sum of the mean time to interrupt (MTTI) and the mean time to repair (MTTR). The mean time to interrupt (MTTI) is the mean time the system or component will be executing before a failure occurs. The mean time to repair (MTTR) is the time it takes to actually repair and recover from the failure; for example, including the time it takes to restore from a checkpoint.

2.1.2 Fault Masking with Redundancy

Fault tolerance is concerned with providing a service conforming to a predetermined specification in spite of faults occurring [63]. A fault tolerant system is one that never produces error state despite the presence of faults. Fault tolerance is typically achieved by fault masking, the ability of the system to *hide* failures and avoid error states [5]. To accomplish this objective, the system leverages redundancy to detect and correct failures. Redundancy is defined as having additional resources that go beyond the minimum needed to complete the required tasks at the expected level of performance. All fault tolerant techniques exploit redundancy to mask failures from the application being supported [63].

There are four forms of redundancy: hardware, software, information, and time [59]. Hardware redundancy is provided by augmenting the system with additional hardware resources to enable system resiliency. An example of hardware redundancy is to execute the same task simultaneously on two different processors, thus making the system resilient despite a single processor failure. Software redundancy, although seldom used, consists of writing the same function using two different methods and then comparing the corresponding output to avoid design faults. Information redundancy, a common technique to provide data resiliency, can be a simple copying a file to multiple locations or could be more elaborate involving complex encoding techniques. Lastly, time redundancy consists of re-executing a failed task, thereby harnessing the time resource much like the other techniques harness hardware, software or storage resource redundancy.

2.1.3 Fault Tolerance Challenges at Exascale

Exascale presents some unique challenges to fault tolerance in exascale systems as faults are no longer an exceptional event. The first challenge is to provide a fault tolerant technique that is efficient when implemented in systems with high failure rates. Additionally, any solution must address how to coordinate all operations across millions, if not billions, of executing threads, support a multitude of different target applications and provide seamless interactions for application engineers. It is unclear if current fault tolerant solutions will continue to be efficient as system sizes grow to meet the demand of future exascale application [38, 92, 10,

13].

Any system implementing fault tolerance can be broken into four distinct phases: detection, containment, recovery, and preparedness [59]. These phases are not unique to HPC systems; but when applied to exascale-class systems, they can pose unique challenges. This section will define these phases and discuss how the challenges they entail are solved in HPC systems.

Fault Detection. To provide fault tolerance, mechanisms must be designed to determine that the system deviated from its expected behavior. Unfortuntely, faults cannot be detected alone, but can only be observed through a fault-induced failure. In other words, rather than detect faults, fault tolerance techniques check for error states resulting from failures. This failure checking process, critical to all fault tolerance techniques, must exhibit the following properties. First, the check should be complete, in the sense that any type of failure should be detectable. Second, the check should be independent of the system being checked, such that a fault in the system will not cause the check to fail. Third, the check should be deterministic, given the system specifications, and should be application independent. In HPC systems, this is typically provided through the Reliability, Availability, and Servicablity, which is a management layer, being run on each individual node within the cluster [34].

To provide fault tolerance, the system must first be able to detect that a fault has occurred. Faults are not possible to detect, but can only be observed through the resulting failure. In other words, the only way to detect faults is to check for error states resulting from failures.

Fault Containment. Once a failure is detected, the fault containment component is responsible for identifying the location of the fault and limiting its effect. For example, in exascale computing fault containment consists of identifying which node produced faulty behavior and removing that node from the system.

Fault Recovery. Fault recovery is concerned with the mechanisms and techniques of reestablishing the expected level of operation of the a system component after a fault has occurred. For checkpointing, this process involves rolling back the system to a known good state and re-executing the code that produced the error state as a result of the failure. This technique assumes that the error state was produced by an operational error in the system

and is not due to a design error in the application code. If an error is a result of a design fault, then re-executing the failing code is bound to produce the same error state. Systems software designs often ignore this possibility; but one can easily see how ignoring design faults may lead to a system that continually re-executes faulty code and therefore never halts. The solution is typically to place a limit on the number of times consecutive fault recovery is performed, such that the probability of operational faults causing consecutive errors is low.

Fault Preparedness. During normal operations, the fault tolerance method is proactively preparing for the occurrence of the next fault. In checkpointing, this is achieved by periodically writing the system state to stable storage. In replication, this process consists of maintaining consistent system state across the replicas. Checkpointing prepares for faults by having a known good state of the application saved in a location that will be accessible, regardless of failure. Replication, on the other hand, must ensure that a consistent state of the system is maintained between the primary and replica processes. For example, non-deterministic functions must be made consistent across all replica processes. Regardless of the fault preparedness method used, the system incurs a potentially significant overhead, even if an error state is never encountered. Fault preparedness is of particular concern in exascale-class systems as the overhead associated with saving global state and maintaining process consistency increases as the size of the system increases.

2.2 ROLLBACK AND RECOVERY PROTOCOLS

Rollback and recovery are the predominate mechanisms deployed to achieve fault tolerance in current HPC environments. In the most general form, the rollback and recovery mechanism involves the periodic saving of the current system state to stable storage, with the anticipation that in the case of a system failure, computation can be restarted from the most recently saved state prior to failure [30, 55]. The identification of an error, before or during a checkpoint, requires that the application rollback to the previously completed checkpoint.

Single node systems can save their system state and be assured that the checkpoint

represents a consistent view of the execution. This is due to the fact that the system is executing based on a single clock, making it possible to easily capture computation at a single point in time [103]. However, distributed computing environments are designed with multiple system clocks. Therefore generating a consistent view of the system state is significantly more complicated than in a single clock system. A large number of solutions have been proposed to achieve consistency in distributed systems. Most of the techniques are derived from the seminal work of Chandy and Lamport [17, 55]. The global-state recording algorithm is typically used, can be simply described as follows: each process records its own state and the state of its communication channels. The question then becomes how to use these states to re-construct a globally consistent view of the application.

2.2.1 Coordinated Checkpoint

The Chandy-Lamport algorithm provides a method for achieving a globally consistent state and guaranteeing that each process's state accurately produce a globally consistent state. To achieve this goal all processes coordinate with one another to produce individual states that satisfy the "happens before" communication relationship [16], which is proven to provide a consistent global state. Essentially, the algorithm provides a method for all processes involved to stop operation "at the same-time" and transfer their system state to a stable storage. This has become known as coordinated checkpointing, whose defining characteristic is that all processes coordinate to write their state to stable storage.

The major benefit of coordinated checkpointing stems from its simplicity and ease of implementation, which lead to its wide adoption in high-performance computing environments. However, its major drawback is lack of scalability, since it requires all processes to coordinate in order to achieve a checkpoint [28, 38, 92, 10, 13].

2.2.2 Coordinated Checkpointing in HPC

Today, most HPC applications implement some form of coordinated checkpointing, which is typically achieved by pausing their execution and requesting the system to capture a checkpoint; this ensures that all running processes have reached the same application execution

point. For example, checkpointing could be done after every 1000 iterations through process loop or based on other easily identifiable stopping-points in the applications execution, referred to as an application barrier. Another approach is to have the system coordinate the checkpoint without the application's knowledge, which may be desirable if process migration by the system software is supported [47, 81].

2.2.3 Uncoordinated Checkpoint and Derivative Work

In contrast to coordinated checkpointing, uncoordinated checkpointing enables processes to record their states independently of each other and postpone creating a globally consistent view until the fault recovery phase [105, 85]. The major advantage of uncoordinated checkpointing is that the overhead during fault free operation can be reduced by allowing processes to checkpoint when its most convenient. For example, a process could produce a checkpoint when the processes state is small [110]. However, uncoordinated checkpointing suffers several disadvantages. First, in order to be able to construct a consistent state during recovery, each process must maintain multiple checkpoints and message logs. Furthermore, it must perform extra work to garbage collect unnecessary checkpoints [109]. Second, uncoordinated checkpoints may be subject to the well-known domino effect, when processes fail to find a consistent state upon failure, which results in the re-execution of the entire application [91].

Communication Induced Checkpointing. One hybrid approach is known as communication induced checkpointing schemes [4, 11]. Based on this technique, processes perform independent checkpoints that ensure a basis for a system-wide consistent state by triggering checkpoints from communication patterns. Although it reduces the coordination overhead, the approach may cause processes to store useless states that are never used in future rollbacks. To address this shortcoming, "forced checkpoints" have been proposed [49]. In most cases, forced checkpoints are used to reduce useless states; it may, however, lead to unpredictable checkpointing rates. Another approach is to increase the likelihood of obtaining a globally consistent state, thereby reducing the number of checkpoints. This approach typically uses a message ordering techniques, such as Lamport clocks [60].

Asynchronous checkpointing. Asynchronous checkpointing with message logging [54,

88], improves checkpointing performance by avoiding synchronization during the checkpoint process, as in uncoordinated checkpointing. In addition to storing system state, there is also a log of all recent messages sent or received by the processes. During the rollback procedure, the message logs can then be used to synchronize the processes. One of the potential advantages of this technique is that nodes can be restored independently avoiding coordinated recovery.

2.2.4 Uncoordinated Checkpointing in HPC

While uncoordinated checkpointing has been well explored in the literature, it is rarely used in high performance computing. The notable exception is FTC-Charm++ which has implemented uncoordinated checkpointing and is still supported in the middleware layer [112]. There are many reasons for the lack of adoption of uncoordinated checkpointing in HPC. The most commonly cited one is the complexity of building and maintaining a globally consistent state from the independent checkpoints from all individual nodes. Another implementation concern is related to the growing size of message logs, as the size of the system increases. There have been attempts to reduce the message log size by exploiting properties of the underlying application. These techniques, however, are dependent upon the application; as such, they have not gained wide acceptance [44].

Although the applicability of uncoordinated checkpoint techniques is still being debated, it is our belief that these techniques are not well-suited to address efficiently the failures expected in future large-scale HPC environments. It has been argued that the sheer number of components in exascale-class systems increases the potential for more frequent faults, making operating under a high likelihood of failure the normal, rather than the the exceptional behavior in these systems. The underlying assumption of fault-tolerance techniques based on uncoordinated checkpointing is that faults are rare events. Although appropriate in computing environments where the fault frequency is low, the intrinsic design decision of uncoordinated checkpoint to delegate synchronization and state consistency to the rollback process is no longer viable in large-scale computing environments, where the propensity to failure is expected to be high. Furthermore, the potentially high number of interrupts due

to the need to perform independent checkpoints at individual nodes may result in cascading system delays, negatively impacting system efficiency and performance.

2.2.5 Reducing Checkpoint Overhead

One of the largest overheads in any checkpointing process is the time necessary to write the checkpoint to stable storage, because while the checkpoint is being written the application must pause its execution. As the systems grow larger, in number of nodes and memory size, the time to write a checkpoint will also increase in order to capture the global system state. The time needed to write the checkpoint is a function of the checkpoint size and the available IO bandwidth, making minimization of the size of the checkpoint a critical concern in exascale [81]. In this section, we will review these optimizations and discuss their possible implications at the exascale level.

2.2.5.1 Incremental Checkpointing Incremental checkpointing [18, 1] attempts to reduce the size of the checkpoint by only writing the system state that has changed since the previous checkpoint. It can be implemented using dirty-bit flags located at the memory page level [86]. Incremental checkpointing can be extended across multiple nodes to use parity and other encoding techniques to further reduce the total checkpoint size [84, 29].

Hash based incremental checkpointing makes use of hashes to determine which state has changed, as opposed to using dirty-bits [1, 18]. The main advantage of this technique is that it allows differences to be detected at a level below the page size, potentially further reducing the size of the checkpoint.

Another proposed scheme, known as copy-on-write protocols [70], offloads the checkpointing process to a secondary task and again only writes incremental checkpoints. This enables the application to continue working while the checkpoint write operation proceeds in the background. To accomplish this the checkpoint sub-system flags memory pages as read-only until they have been written to stable storage. In the even that the application needs to write to that page then an in-memory copy will occur such that the application can write to the page but the previous state of the page is retained until copied. The concern

with this technique is that some applications would require double the memory to support the simultaneous execution of the checkpoint and the application.

2.2.5.2 High-speed Local Storage Traditionally, HPC nodes are not configured with local storage, such as spinning disks or solid state devices, instead they network boot and their root filesystem is maintained on network attached storage devices. The network then becomes a bottleneck in the process of writing checkpoints because the only stable storage device is the network attached storage. It has been suggested that nodes in exascale systems should be configured with fast local storage and that doing so could reduce checkpoint write times to between four minutes and one second [2]. This would improve performance of checkpointing such that exascale-class systems would be 59-97% efficient, which while not ideal, might be sufficient.

There are two problems with this approach: increased failure rates of individual nodes and increased per-node cost. Historically local storage dramatically increased the node failure rates. The use of solid-state devices, as opposed to spinning disks, has been proposed to address this shortcoming. However, durability of these devices when operating in an environment with a high volume of writes can become an issue. Further, the relatively large cost of the these storage devices can prohibitive with regard to the overall system costs [19].

One proposed method for using this high-speed local storage, should it be available, is multi-level checkpointing, which consists of writing checkpoints to multiple storage targets [79]. For example, a checkpoint is first written locally, then to a neighbor node and ultimately to a parallel filesystem. Local storage is faster than writing to the parallel filesystem directly. This enables the application to return to normal execution faster, effectively reducing the checkpoint write time. This technique has been shown, through analytical models, to significantly reduce the overhead of checkpointing in exascale-class systems [99, 46]. The disadvantages are that it complicates the checkpoint writing process and requires that the system track the current location of all process's checkpoints.

2.3 STATE MACHINE REPLICATION

Process replication and state machine replication have long been used to provide fault tolerance in distributed [9, 61, 100] and mission critical systems [7]. State machine replication creates one or more replicas for each process and ensures that all replicas deterministically calculates the output in response the given inputs. To ensure consistent system state, all process messages must be delivered to all nodes running a given process, typically done by using a message ordering protocol [62]. Additionally, if any parts of the computation rely on nondeterministic calculations, such as random number generation, then extra consistence protocols must be constructed. In the event that two processes produce different results, a voting protocol is used to determine the correct outcome.

The advantage of state machine replication is that it provides a fault tolerance method that can mask most failures without the need to re-execute computations. Additionally, replication can be used to detect and correct system failures that are otherwise undetectable, such as silent data failures [80] and Byzantine faults [73, 14, 40, 108]. However, replication alone is not enough to guarantee fault tolerance since it is possible that all nodes executing a given process could fail simultaneously, thus replication is typically paired with some form of checkpointing.

2.3.1 Replication in HPC Systems

Replication been proposed as a viable alternative to checkpointing in high-performance applications [38, 92, 10, 13, 68]. Previously it was thought that replication was too costly to work efficiently in HPC [33], however the predicted increase in failure rates in future systems has re-ignited interest in this technique. As discussed in Section 1.1.2, it is hypothesized that in exascale-class systems efficiency of existing checkpointing techniques could drop below 50% making it equal to that of state machine replication. Figure 2 presents the percentage of the execution actually executing applicaiton code versus the time performing coorindated checkpoint or recovering as the system size grows that expected in exascale-class systems. The figure assumes a fixed the checkpoint time of 20 minutes. Twenty minutes is an estimate

of checkpoint time given the amount of system memory (32-64PB) and the I/O bandwidth (20-60TB) projected in the "swim-lanes" in Table 1, producing a checkpoint write time of between 9-54 minutes, which is inline with previous studies [12, 37].

Replication has been proposed as an augmentation to existing checkpointing techniques as a means to reduce the checkpoint interval in HPC, while maintaining the same system resiliency levels [38, 13]. To limit the overhead of replication there has been proposals to for partial replication [104, 26]. Replication has also been used in HPC to guard against silent data corruption [80]. There are several different implementations of replication in the widely used MPI library, each with their different tradeoffs and overheads. The overhead can be negligible or up to 70% depending upon the communication patterns of the application [32].

2.4 POWER MANAGEMENT

Power management is a mature research area and several energy saving runtime techniques have been proposed [42, 50, 51, 56, 69, 72, 94, 106]. This work spans real-time systems, mobile devices and data centers and they all revolve around the notion that one can save energy by leveraging execution slack. In real-time systems the interplay between slack, scheduling and global energy savings have been extensively studied [113, 53, 6]. In mobile devices the operating system is tasked with properly managing the devices sensors and network accesses to maximize its effective lifetime [95, 107]. In data centers the focus is on reducing energy consumption of the entire ecosystem, from reduction in cooling to peak power management.

Much of this work relies upon some form of DVFS, which is a mechanism by which the frequency and voltage of a processor can be scaled during CPU operation. DVFS [83] and clock throttling [78] are attractive since CPU power continues to dominate overall system power consumption [50]. Dynamic CPU power, P, can be determined by knowing the chip activity factor, A, the capacitance C, operating voltage, V, and the frequency, f. The dynamic CPU power is therefore represented by the function $P(A, C, V, f) = A \times C \times V^n \times f$, where $n \geq 2$. DVFS enables both voltage and frequency to be change. The reduction in both the frequency and voltage of a processor has a near cubic relation to the amount of power

consumption of a CPU. The argument for energy savings is that you achieve a polynomial reduction in power consumption at the cost of a linear increase in execution time. This is not a perfect argument because the linear slow down assumption is highly dependent upon the type of application.

2.4.1 Power Management in HPC

Power management in data centers is most closely aligned with the demands of the high performance computing environments; in fact, much of the work overlaps. There are two areas in which energy and power for supercomputing is an issue. The first is energy consumption, which directly equates to the cost of running the system [41, 64, 43]. The second is peak power usage, which can constrain a system, keeping it from running at full speed to limits on the rate at which energy can be consumed [111]. There has been work that has looked specifically at the effect of DVFS has upon HPC workloads, concluded that there is possible energy savings but it is highly dependent upon the application [64, 43].

2.5 FAULT TOLERANCE AND ENERGY CONSUMPTION

Fault tolerance and power management have been studied extensively, although only recently have researchers begun to study the combination of these two competing goals. Both fault tolerance and power management seek to exploit slack in the system, both in time and power, to provide fault free computation or energy savings respectively. There is also a delicate interplay between these two fields, for example replication and rollback techniques requires additional energy to accomplish their goals. Conversely, it has been shown that lowering supply voltages, which is often done to conserve energy, causes an increase in the probability of transient faults [15, 48, 115]. The tradeoffs between fault free operation and optimal energy consumption is only now being explored and high performance computing is a fertile ground for such research.

In the real-time embedded systems there has been a number of studies that combine the

goals of fault tolerance using DVFS. Studies have paired scheduling techniques and power management to simultaneously solve real-time deadlines in-spite of errors and optimize energy consumption. The most relevant work uses of DVFS to make triple modular redundancy (TMR) more energy efficient [114, 27]. This work develops an analytical model to describe the potential energy savings of reducing the execution speed of the third replica, referred to as optimistic TMR (OTMR). More recently work on reliability-aware power management (RAPM) explicitly explores the tradeoffs and proposes frameworks for characterizing this interplay [89, 90].

The energy consumption of HPC systems during checkpoints was studied in [24] for a variety of checkpointing methods. Models have been developed [74] for coordinated, uncoordinated (message logging) and parallel recovery methods. However, the assessments did not consider the effect DVFS might have upon their energy consumption. While the power profile of several systems is understood during checkpointing, these studies did not look at modifications to save energy. The one exception is a study of *local* checkpoint operations that used DVFS to achieve energy savings [96]. Work in this thesis analyzes explicitly the use of DVFS during both local *and* remote checkpoint operations. Others have exploited the causal relationship between CPU temperature and reliability to build an adaptive fault tolerance approach that mitigates faults by reducing the CPU power consumption, which has the side effect of also conserving energy [98].

2.6 SHADOW REPLICATION IN CONTEXT OF RELATED WORK

The work in this thesis lies at the intersection of the three presented disciplines: fault tolerance, power management and high performance computing. The existing work in this area has largely been focused upon measuring power and energy consumption of existing fault tolerance techniques and only a few have proposed new techniques [96, 24, 74]. This thesis proposes a new fault tolerance method, *Shadow Replication*, which can achieve faster time to solutions and decreases in energy consumption in power-constrained environments.

To achieve fault tolerance redundancy is required, in this case time or hardware redun-

dancy. The amount of redundancy is a factor of the minimum required time or hardware, represented as T_{min} and H_{min}. For example, assuming only one error, re-execution requires a maximum of double the time, whereas traditional replication requires double the hardware resources. However, depending upon the number of failures and the degree of replication, redundancy factors could be more than double. The redundancy factor of different fault tolerant methods are represented in Figure 3. As depicted, shadow replication is designed to provide a framework to better balance redundant resources to provide more energy efficient fault tolerance in high performance computing. *Shadow Replication* achieves this by enabling a tradeoff between hardware and time redundancy.

Using DVFS to reduce the execution speed of the replica process, one reduces the hardware resources while potentially increasing the necessary time. The computational model of *Shadow Replication* thereby enables the parameterized tradeoff between these two redundant resources: hardware and time. Energy is a function of power consumption and length of time, in terms of redundant resources, the energy is determined by the hardware requirements and the time it takes to successfully execute the application. This thesis develops models to leverage the tradeoff enabled by *Shadow Replication* and demonstrates that this fault tolerance model has potential to reduce energy consumption in exascale-class computing environments while meeting expected levels of performance.

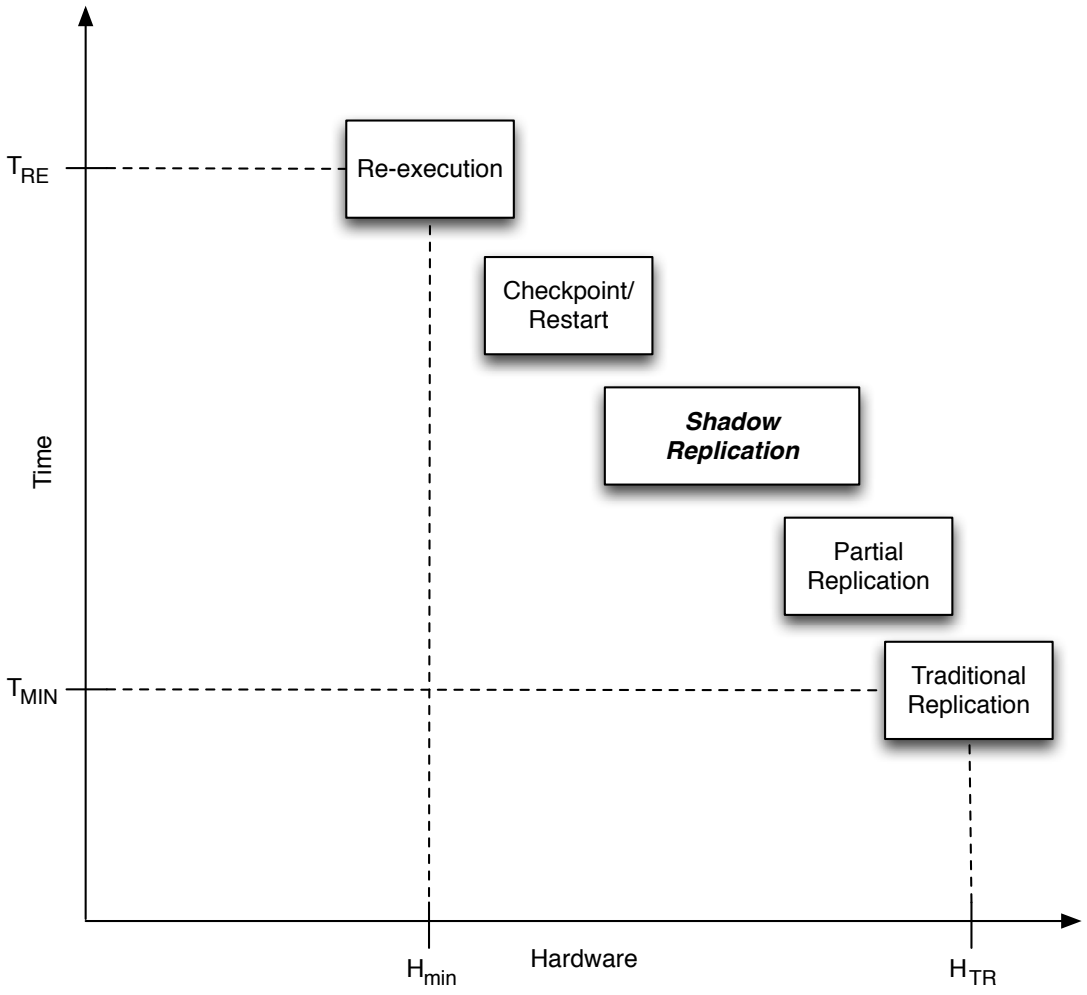


Figure 3: The use of time and hardware redundancy to enable fault tolerance methods, with a focus on how *Shadow Replication* provides a framework for balancing these resources.

3.0 SHADOW REPLICATION

Replication has been proposed as an alternative to checkpoint/restart protocols; it has been shown to provide significant improvement in *scalability*, with the potential to reach the efficiency levels at scales required by extreme scale systems. The advantage of replication is that its implementation requires neither new hardware features nor modification to the application software. However, these techniques have not gained wide deployment in HPC environments. This is mostly due to the need for increased levels of hardware, energy and power requirements. To realize fault tolerance and ensure levels of performance similar those achieved when no failures occur in the original infrastructure. Shadow replication attempts to address this shortcoming and aims to provide the required levels of fault-tolerance while minimizing energy in HPC environments. This chapter defines the *shadow replication* computational model and describes it's dynamics in Section 3.1. Section 3.2 restricts the general computational model into an execution model applicable in the high performance computing environment. Using this execution model, Section 3.3 defines an analytical model for the expected energy consumption of shadow replication and other fault tolerant techniques. Using this analytical model we then formulate the optimization framework for determining optimal execution speeds for *shadow replication*.

3.1 SHADOW REPLICATION DETAILS

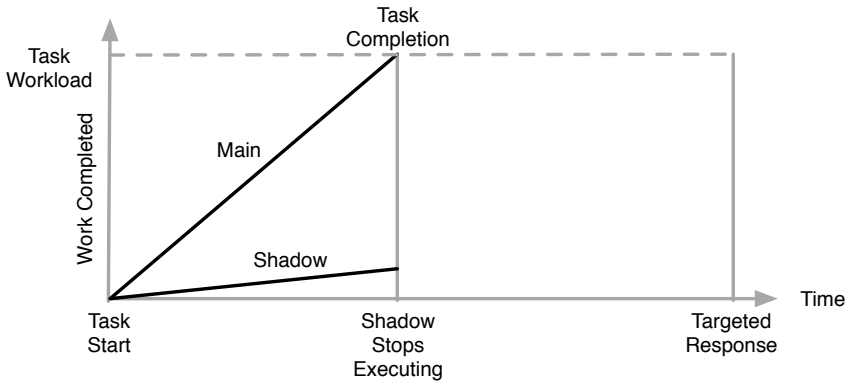
Current fault tolerance approaches rely upon either time or hardware redundancy in order to tolerate failure. The first approach, which uses time redundancy, requires the re-execution of work after the failure is detected. Although it can be improved by the use of

checkpoint/restart systems, such an approach still requires time redundancy. The other approach, replication, which relies upon hardware redundancy, requires both hardware, power, and energy to execute the task twice. Shadow replication, by contrast, seeks to achieve the optimal tradeoff between re-execution and traditional replication to meet the constraints of the application, while minimizing hardware and energy resources, subject to the system power consumption constraints.

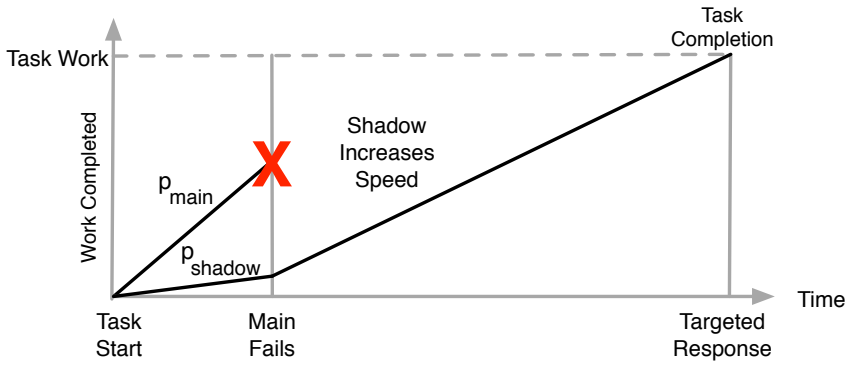
Exploring the energy consumption of replication reveals that the replica processes could execute at different execution speeds while still guaranteeing a response time, not to exceed that provided by checkpointing. Based upon this observation, the basic idea of shadow replication is to associate with each main process a "shadow process," whose execution speed depends on the performance requirements of the underlying application and the likelihood of failure. In order to overcome failure, the shadow executes concurrently with, but on a separate computing node than the main process. The successful completion of the main process results in the immediate termination of the shadow process. If the main process fails, the shadow process takes over the role of the main process and resumes computation – possibly at an increased speed – in order to complete the task within the targeted response time. The dynamics of shadow replication are illustrated in Figure 4(a).

Depending on the occurrence of failure, two scenarios are possible. The first scenario, depicted in Figure 4(a), takes place when no failure occurs. In this scenario, the main process executes at the speed necessary to achieve the desired level of fault-tolerance, minimize power consumption and meet the target response time of the supported application. Based on the specified execution speed, the main process completes the total amount of work required by the underlying application. During the execution time of the main process, the shadow executing at a reduced processor speed, only completes a significantly small amount of the original work, before it is terminated.

The second scenario, depicted in Figure 4(b), takes place when failure of the main process occurs. Upon failure detection, the shadow process increases its processor speed and executes until completion of the task. The processor speed at which the shadow executes after failure is determined so that the task completes by the targeted response time.



(a) Case of no failure



(b) Case of failure

Figure 4: Shadow replication computational model.

Two properties are unique characteristics of the shadow replication computational model:

- Given that the likelihood of individual socket failure is low in exascale systems, the first execution scenario of shadow replication is likely to occur more frequently during the execution of an application. Consequently, the additional energy consumed by the shadow is significantly smaller than the energy needed to execute an exact replica of the main, resulting in significant energy savings.
- The failure of a shadow has no bearing on the behavior of the main process

Early detection of the main process failure is an important aspect of shadow replication to ensure optimal execution. Several mechanisms have been proposed to achieve early failure detection, a challenging problem in distributed systems [59]. These techniques differ in their ability to detect failures as soon as it occurs and the overhead required to do so. It is worth noting, however, that failure detection in shadow replication is limited to the main and its shadow and does not involve other processes in the system. A simple mechanism to detect failure is to use the estimated execution time of the main process as an indicator of successful completion or failure of the main process. If the shadow process does not receive a notification of a successful completion, the shadow assumes failure of the main process and proceeds to complete the task. Although the overhead is low, this simple method runs the risk of causing the shadow to wait longer than necessary, therefore causing the shadow to execute sub-optimally. This simple solution can be easily augmented with a light-weight heart-beat protocol to achieve earlier failure detection while reducing energy consumption.

Another critical issue of shadow replication is the need to determine the speed at which the main must execute to increase the likelihood of successful completing the task, when a failure occurs. On one hand, increasing the execution speed of the main process leads to quicker completion time, which in turn reduces both the shadow execution time and exposure of the main process to failure, thereby achieving significant energy savings. Furthermore, this strategy allows the detection of failure, if such an event were to occur, early in the execution phase, which provides additional laxity for the shadow to complete the task by the targeted response time. On the other hand, increasing the processor speed of the main processes may stress the system beyond its power limits which is likely to have an adverse

impact on the resilience of the system.

The approach used in this thesis to obtain optimal execution speeds for the main process and its associated shadow takes into consideration the nature and computational attributes of an HPC application, without stressing the system beyond its thermal and power limits. In the following sections a brief discussion is provided to highlight how the shadow replication computational model is implemented in a distributed system; further details are presented in Chapter 6, which discusses the implementation of shadow replication in the MPI middleware commonly used in HPC environments.

3.1.1 Process Mapping

In shadow replication, the execution of a task spawns the creation of both a main process and a shadow process. These processes must be carefully mapped to the computing nodes[1] of the distributed infrastructure to achieve fault tolerance. Specifically, the mapping must be done such that the main and shadow processes are *fault-isolated* from each other, meaning that a fault affecting one process does not affect the other. Fault isolation is necessary to minimize the likelihood that both the main and shadow processes fail at the same time.

There are a number of different ways to achieve a fault-isolated mapping of shadow replication processes. One possible strategy is to make use of the multi-core capabilities of the infrastructure to assign main processes and shadows such that a given shadow process can only be run concurrently with an unrelated main process. In such a mapping, voltage scaling is used to execute the main and the unrelated shadow process at their specified execution speeds. Figure 5 illustrates a feasible assignment using this approach for the case of three main processes and their associated shadows. Another strategy is to use time-sharing to achieve both process mapping and process slow-down without the use of voltage scaling, only the first strategy is considered in this thesis. The MPI implementation of process mapping will be explored in further detail in Chapter 6.

[1] In HPC computing nodes are the sockets which are performing the computation.

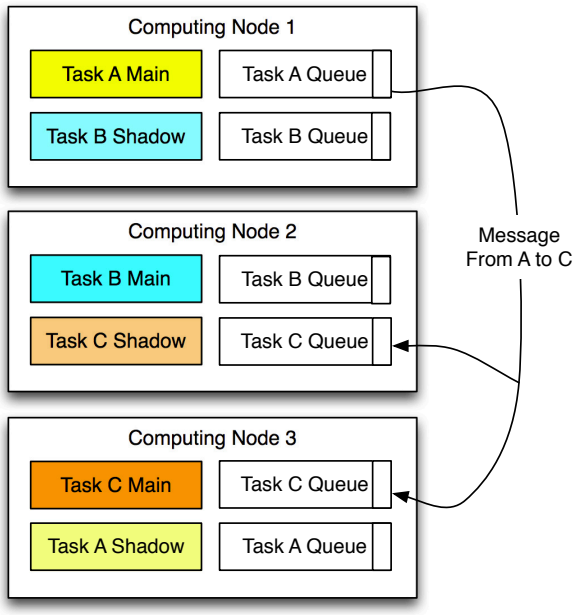


Figure 5: Example Process Mapping

3.1.2 Message Passing Considerations

Another important aspect of the shadow replication model is providing intra-process communication to achieve synchronization and to maintain system consistency. Any communication model must, at a minimum, provide the following two properties:

- All messages destined for a task must be delivered to both the main process and the associated shadow process.
- It's possible that both main and shadow processes send the same, and duplication of the same message must be resolved.

To satisfy the communication and synchronization requirements of the shadow replication model, the runtime support environment makes use of message queues at each replica, possibly consolidating these queues between a replica-pair. The *replica-pair* is the unit consisting of a main and it's associated shadow process.

It is clear that the policy of directly forwarding messages to all processes is not likely to scale in exascale-class systems. The use of *passive-queues*, where stored messages are only forwarded upon request by the processes, has greater potential to achieve the levels of scalability expected in exascale-class systems. This approach eliminates the need for the queue manager to notify processes directly; it also allows processes to request messages when they are ready. Consequently, processes running at a higher execution speed will not interfere with the execution of slower running processes. As will be discussed in Chapter 6, a passive-queue for communication messages is well-suited for MPI-based environments, where a similar data structure is already supported by the communication middleware layer.

In order to keep track of which replicas are present, a global register, which maps processes to the various replicas, is needed. In our MPI implementation this is achieved by using a global process map, a static map constructed when MPI initializes the nodes. A detailed discussion of these maps is provided in Chapter 6.

When a message is sent, the global register is consulted and the message is routed to each of the replica queues. The queue holds the message until it has been delivered to all associated processes[2]. This is possible because all associated processes are registered in the process map. An example of passive-queue based message delivery is depicted in Figure 6. While not shown in the figure, messages would also be removed from the queue once the main process has completed. This scheme ensures that all executing processes reliably receive all messages destined for their task.

A duplicate-free implementation of message queues maybe achieved by ignoring messages that have already been received. More specifically, when the queue receives a message from a task, it determines if that message has already been received by the queue. If the message is redundant, it simply ignores the message. However, if it is new, the message is queued for delivery. An example of the message receiving process is shown in Figure 7. This allows shadow processes to execute at their intended speeds, without producing duplicate system messages. An additional benefit of this model is that messages will be processed regardless of their source; therefore, the queues need not be aware of process failures.

[2]Any implementation of such a system must address the issue of growing queue size. More detailed discussion in Section 4.1.4

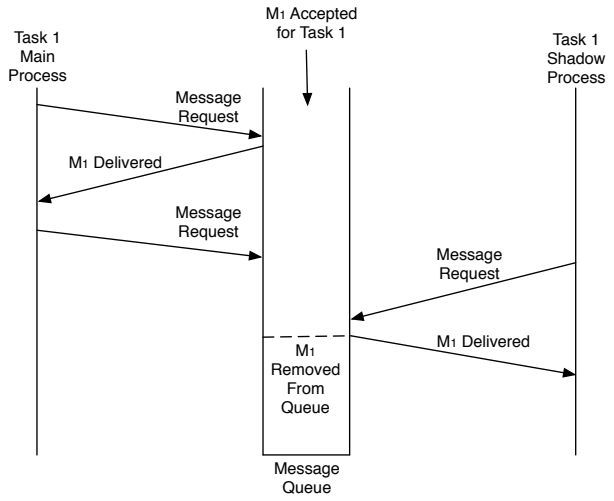


Figure 6: Example Message Delivery

3.2 SHADOW REPLICATION EXECUTION MODEL DEFINITION AND OPTIMIZATION

To study the application of shadow replication and the potential energy savings, this section defines an execution model for HPC environments, the potential optimizations, and the supported applications. The execution model enables us to explore the possible optimization points. Pairing these optimizations with a supported application provides the information necessary to both develop and evaluate analytical models and simulators representing shadow replication.

3.2.1 Execution Model

We consider applications executing in a distributed computing environment using a large number of collaborative tasks (equivalent to ranks in MPI). The successful execution of the application depends on the successful completion of all tasks. Therefore, the delay of a single task delays the entire application.

The execution model of shadow replication explored in this thesis consists of two processes

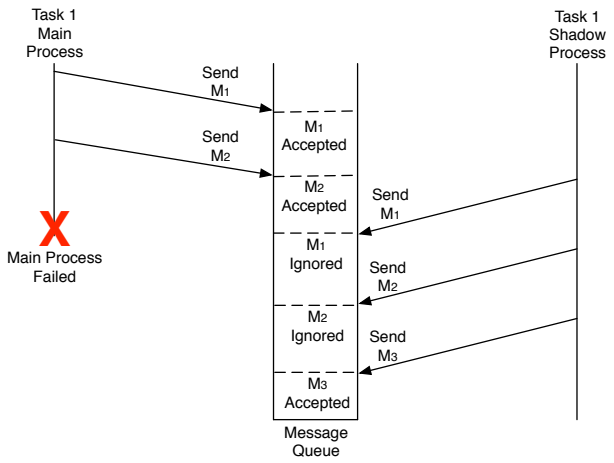


Figure 7: Example Message Receiving

for each task: a main and a shadow. The main process executes at a single execution speed denoted as σ_m. If no failure occurs, the task will be completed by the main process. The shadow process executes at two different speeds, a speed before failure detection, σ_b, and a speed after failure detection, σ_a. If a failure occurs in the main process, the shadow takes over the execution and changes its execution speed to σ_a. If a failure occurs in the shadow process, the main process simply continues to execute at σ_m. This is depicted in Figure 8.

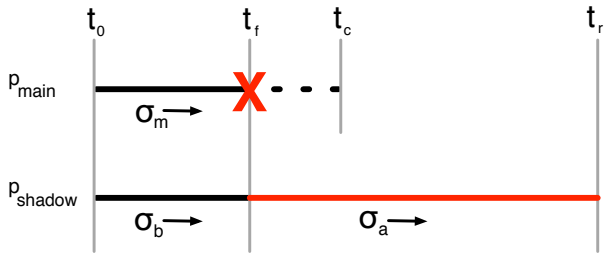


Figure 8: Overview of Shadow Replication

Additionally, Figure 8 defines time points signaling system events. The time at which the main process completes a task is t_c. The time of failure in the main process is denoted as t_f. The time at which the shadow process is guaranteed to complete a task, regardless of

a failure, is t_r. There is a longer discussion about how this guarantee is determined later in this section.

The challenge is determining execution speeds such that the application requirements are met while simultaneously maintaining the system-level goals. When applying shadow replication to exascale, the system-level goals are to minimize energy, power, and/ or time to solution. These goals maybe conflicting; for example, increasing power consumption might decrease the time-to-solution but have no effect on energy consumption. The next section explores the possible design choices and discusses possible optimization strategies to achieve these goals.

3.2.2 Shadow Replication Optimization Issues and Strategies

Regardless of the minimization objective, one can explore the optimization search space to gain a better understanding of the issues related to the design and use of shadow replication.

The execution speeds of the main and its associated shadow are the variables of the optimization model, and their optimal values are the outcome of the model. These variables determine the task completion time and the power and energy consumption. The remaining parameters capture the computing capabilities of the system and the performance requirements of the supported applications. The system parameters, such as number of nodes, maximum execution speed, node failure, and power constraints; capture the intrinsic characteristics of the system that are relevant in computing the optimum values of σ_m, σ_b, and σ_a. The application parameters constrain the optimization model to produce optimum values for the speed variables that adhere to the performance requirements of the application. The resulting optimization model is depicted in Figure 9.

In its most general form, the optimization model seeks to optimize all speed variables. Such an approach, however, may increase unnecessarily the complexity of the optimization problem, particularly when the characteristics of the computation environment and application may constrain the formulation of the optimization problem. Such insight must be taken into consideration when deriving the optimum values that are relevant to and commensurate with the environment supporting the application. For example, in HPC environments,

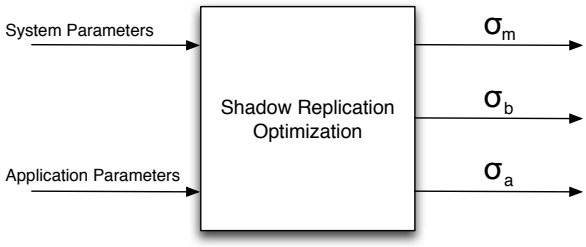


Figure 9: Optimization Model

the high cost of the computing infrastructure, coupled with the need to reduce the time to solution makes it undesirable or even unacceptable to execute the main process at less than the maximum speed. The reality of such an environment dictates that σ_m be set to the maximum execution speed. A summary of all possible optimization strategies, along with the type of application for which the strategy is well-suited, is depicted in Table 2.

When calculating any optimization one can perform the optimization either online or offline. In this case, it is always most logical to calculate σ_m and σ_b offline because the execution speeds are needed at the beginning of execution, prior to any other system event has occurred. However, it is possible to calculate the execution speed of the shadow after failure either before the failure occurs (offline) or once a failure has been experienced (online). Depending upon the information known and the objective of the optimization, the offline optimization could perform as well as the online calculation, making online optimization unnecessary.

Lastly, one needs to define the objective of the optimization. In this thesis, our goal is to minimize the energy consumption while achieving high throughput. Although using this execution model, one could also find execution speeds which minimize power consumption or time to solution. When performing an online optimization of the speed of the shadow after failure, it might be unnecessary to optimize energy and instead optimize power or time to solution.

σ_m	σ_b	σ_a	Explanation or Example
Opt.	Opt.	Opt.	Full optimization
Opt.	Opt.	Fixed	Always execute at max speed after failure or we could let $\sigma_a = \sigma_b$.
Opt.	Fixed	Opt.	Always execute the shadow at 0, effectively making this model act like re-execution.
Opt.	Fixed	Fixed	Execute at speed 0 before failure and maximum speed after failure, only optimizing the speed of the main process.
Fixed	Opt.	Opt.	Always execution the main at max speed, only optimizing the shadow execution speeds.
Fixed	Opt.	Fixed	Execute the main and the shadow after failure at maximum speed.
Fixed	Fixed	Opt.	An example would be to always execute main at max speed and shadow before failure at zero, only optimizing the shadow after failure has occurred.
Fixed	Fixed	Fixed	This would be the case if we used traditional replication, in which case all speeds are set to the maximum execution speed.

Table 2: Optimization Search Space. The models and simulation we present in future chapters will always fall into one of these categories.

3.2.3 Application Specification

We consider applications executing in a distributed computing environment using a large number of collaborative tasks (equivalent to ranks in MPI). The successful execution of the application depends on the successful completion of all tasks. Therefore, the delay of a single task delays the entire application.

One of the key application factors is the amount of coordination between the tasks. For example, an application might require no intra-task communication, allowing a single task to fail and restart without effecting the completion time of other tasks. However, if intra-task communication is required, then the failing of a single task could potentially delay all tasks with which it would have normally communicated. The amount of communication can be thought of as the amount of dependencies one task has upon all other tasks and affects the application completion time and energy consumption. There are wide range of potential applications but they can be classified into one of the following types: No Dependency, Blocking Dependency or Full Dependency. Details of these communication types can be found in Table 3.

Application Type	Case	Description
No Dependency	Map Tasks	There are no intra-task dependencies, all tasks must complete but could do so with no coordination.
Blocking Dependency	Independent Simulation	Tasks can execute independently but are required to communicate at the end of execution to complete the solution. This represents applications that divide work into non-overlapping parts but require collective operations at the end of execution or perform asynchronous communication during execution, the difference being how much work is necessary at the end of execution.
Full Dependency	Coordinated Simulation	Tasks are fully-dependent upon other tasks and require constant communication between tasks. This represents an application that frequently performs collective operations during execution.

Table 3: Classification of application communication types and their descriptions.

We assume each application is fully parallelizable, with a total workload of W. The work is assumed to be evenly divided into N tasks. We further assume a strong scaling application; therefore, as the number of tasks increases, the amount of work each individual task performs decreases linearly with the number of tasks. Each task is assigned an equal workload, $W_{task} = \frac{W}{N}$, where N represented the number of tasks. Blocking Dependency and Full Dependency applications require N processes to execute in parallel in order to complete an application with N tasks. However, applications that do not require intra-task communication may execute their tasks in a serial fashion.

We will assume that each process is assigned a processing socket to complete its execution. Work is defined as a number of clock cycles and each computing socket has a variable speed, σ, given in clock cycles per second. Therefore, the minimum solution time for an application executing in parallel occurs when all sockets are executing at maximum speed is $t_{min} = \frac{W_{task}}{\sigma_{max}}$. If tasks exhibit no intra-task communication then they can execute in serial, making the minimum completion time depend upon the number of sockets available, $sock_{avail}$, and can be expressed as $t_{min} = \frac{W_{task}}{\sigma_{max}} \frac{N}{sock_{avail}}$.

Each task also has an associated targeted response time, t_{resp}, which is the maximum time that the process needs to complete its task. We express the targeted response time as a multiplicative function of the minimum response time, $t_{resp} = \alpha \times t_{min}$, where α represents a laxity factor defined by the application. For example, if the minimum response time is

100 seconds and the targeted response time is 125 seconds, the laxity factor is 1.25. In contrast, checkpointing techniques assume that if a single failure occurs, the system must always have enough time to re-execute. In this framework this results in a laxity factor of $\alpha = 2.0$, assuming a single failure. If multiple failures occur, checkpointing may require a laxity factor of $\alpha > 2.0$. Such a high laxity is needed, as the application tasks may have to restart multiple times.

3.3 ENERGY CONSUMPTION ANALYTICAL FRAMEWORK

This section develops the analytical framework used to model the energy consumption of fault tolerance mechanisms; specifically, shadow replication, traditional replication, and checkpointing. To calculate the energy consumption, we begin by modeling the power consumption of a process executing on a socket at a given execution speed. In order to model the energy consumption of the entire system, the single socket model is combined to represent a group of sockets. By then combining this with a socket failure model, one can derive the expected energy consumption of various fault tolerance mechanisms, including shadow replication.

3.3.1 Power Model

We start by describing a power model for a single computing socket which will be built upon for our checkpoint and replication models. Consider the dynamic CPU power which is known to be affected by the execution speed of the processor. Specifically, one can reduce the dynamic CPU power consumption at least quadratically by reducing the execution speed linearly. Dynamic CPU power, P, can be determined by knowing the chip activity factor, A, the capacitance C, operating voltage, V, and the frequency, f. The dynamic CPU power is therefore represented by the function $P(A, C, V, f) = A \times C \times V^n \times f$, where $n \geq 2$. DVFS scales both voltage and frequency whereas activity and capacitance are fixed, therefore we denote the execution speed using σ, which is the the combination of V and f. Thereby,

allowing us to represent the power consumption as the function $P(\sigma) = \sigma^n$, resulting in a polynomial function where $n \geq 2$. In the remainder of this thesis we assume that the dynamic power function is the cubic, $P(\sigma) = \sigma^3$.

Next, consider the "overhead" power which is consumed regardless of the speed of the processor. This includes both CPU static leakage and all other components consuming power during execution (memory, network, etc.). In this work, we define the overhead power to be a fixed factor, ρ, of the power consumed when the CPU is operating at full speed. The percentage of overhead power in a system is thus defined as $\frac{\rho}{\rho+1}$. By reducing the execution speed, one can only change the dynamic power and the overhead power remains constant.For example, when ρ equals 4.0 and the CPU is executing at maximum speed, the overhead power of the system is 80%, meaning that the CPU is consuming the remaining 20% of the total power.

The energy consumed over a period of time is the summation of the instantaneous power consumed over that period. Therefore, the energy consumed by a socket executing at speed σ during an interval $[t_1, t_2]$ is given by:

$$E_{soc}(\sigma, [t_1, t_2]) = \int_{t=t_1}^{t_2} (\sigma^3 + \rho\sigma_{max}^3)dt = (\sigma^3 + \rho\sigma_{max}^3)(t_2 - t_1) \tag{3.1}$$

3.3.2 Failure Model

A failure can occur at any point during the execution of the main task, rendering the work completed by that process unrecoverable. Because tasks execute on different computing nodes, failures are assumed to be independent events. However, it is assumed that either the main or the shadow can fail, but not both. If the main task fails it is implied that the shadow will complete without failure. This assumption is realistic as the probability of both the main and shadow sockets failing simultaneously is highly unlikely, given that the failure of any socket is very low. In order to achieve higher resiliency, one could make use of multiple shadow processes to overcome simultaneous failure of the main and its associated shadow.

We further assume the existence of a probability density function, $f(t)$, to express the probability of the main task failing at time t. In the remainder of this thesis, we use an exponential probability density function, thus $f(t) = \frac{1}{M_{soc}} e^{-t/M_{soc}}$, where M_{soc} is the socket

MTBF. We choose to use the exponential distribution function because it is widely used to represent socket failures in HPC environments. This further enables the integration of existing failure models, such as Daly's checkpointing approximation model [23], into the analysis and experimental studies carried out in this thesis.

3.3.3 Replica-Pair Energy Model

This section develops a model which represents the energy consumption of a replica-pair. This model is then used to determine the energy consumption of the combination of replication and checkpointing.

A replica-pair consists of two process, the main and the shadow. The main process executes at a single execution speed denoted as σ_m. If no failure occurs, then the task will be completed by the main process. The shadow process executes at two different speeds: namely before failure detection, σ_b and after failure detection, σ_a, as depicted in Figure 8.

We define some specific time points signaling system events. The time at which the main process completes a task, t_c, is given as $t_c = W_{task}/\sigma_m$. Note that t_c is dependent upon the execution speed of the main. Additionally, we define t_f as the time at which a failure in the main process is detected. Without loss of generality, in order to make the optimization problem formulation easier to understand, we assume $t_f = t_c$, when no failure occurs. We further define the time at which the shadow process completes a task, regardless of a failure as, $t_r = t_f + (W_{task} - \sigma_b t_f)/\sigma_a$. In order to meet the performance requirements of the application, we constrain this value to be less than or equal to the targeted response time of a task, t_{resp}.

We define the expected energy of a shadow replica-pair as the summation of the expected energy consumed by the main and shadow process given our failure model. We assume at most one failure between the main or the shadow process. Based on these assumptions, the

expected energy consumption of a main and shadow replica-pair can be expressed as:

$$\begin{aligned} E_{soc_{rep}} = & \int_{t=0}^{t_c} (E_{soc}(\sigma_m, [0,t]) + E_{soc}(\sigma_b, [0,t])) f(t) dt \\ & + \int_{t=0}^{t_c} E_{soc}(\sigma_a, [t, t_r]) f(t) dt \\ & + (1 - \int_{t=0}^{t_c} f(t) dt)(E_{soc}(\sigma_m, [0, t_c]) + E_{soc}(\sigma_b, [0, t_c])) \end{aligned} \tag{3.2}$$

In the above expression, the first part of this equation represents the expected energy consumed by the main and shadow process before a failure occurs in the main process. This is the summation of the expected energy consumed by the main plus the energy consumed by the shadow given our failure model over the total duration, 0 to t_c. The second part of this equation is the expected energy consumed by the shadow after failure occurs. The duration of this is from the time of failure, t_f, until the shadow completes execution, t_r. The last part is the expected energy consumed by the main and shadow processes in the event that no failure occurs. This equation can then be used as an objective function in the construction of an optimization problem used to find energy-minimizing execution speeds.

If there is blocking or full dependency among the tasks, sockets must continue to execute until the last task completes. To account for energy consumed while waiting, the model calculates the expected waiting time and energy consumption for the replica-pair. Adding the expected waiting energy to the overall expected energy consumption expands the model to account for different application communication patterns.

The energy consumed by the processes waiting is dependent upon two variables: the execution speed of the waiting sockets, σ_{wait}, and the completion time of the last executing socket. The execution speed of the waiting socket depends upon the type of communication. In the case of blocking communication, sockets wait idly until the last task fully completes, and all waiting tasks will have an execution speed of zero, but will still consume overhead power. If, however, the application requires full communication, then tasks will have to perform work while waiting on the last task to complete. In the worst case scenario, the tasks may have to execute a maximum speed while waiting. The execution speed of the waiting socket, σ_{wait}, can then be either zero or σ_{max}, depending on the communication type of the application. In the absence of failure, no task is required to wait. If at least one

process fails, however, all other processes must wait until the shadow of the failing process completes its task. Consequently, the expected waiting time depends on the probability, p_{fail}, that at least one main process fails. The expected value is calculated by determining the probability that none of the main tasks would fail, then using this value to derive the probability that one would fail.

$$p_{fail} = 1 - (1 - \int_0^{t_c} f(t)dt)^{N/2} \tag{3.3}$$

Note that because of replication at most $N/2$ nodes, representing the main processes, may fail.

The tasks have to wait until the last task completes its work, which is dependent upon the time of failure and the execution speed of the shadow after failure. Assuming that a failure of the main process has occurred, with the probability p_{fail}, the failure time can be estimated using a truncated exponential distribution. The distribution is truncated to only include the time of completion of the main process, t_c. The average time to failure occurs at the mean of the exponential distribution, truncated at t_c, which is given by the following equation [3].

$$\hat{t_f} = \frac{1}{\lambda} - t_c \frac{1}{e^{\lambda t_c} - 1} \tag{3.4}$$

Using this value we can determine the expected waiting time given the execution speed of the shadow after failure.

$$t_{wait} = \frac{W_{task} - (\sigma_b * \hat{t_f})}{\sigma_a} \tag{3.5}$$

The energy consumed by the waiting processes depends upon the execution speed of the waiting process, σ_{wait}, the failure of the shadow and main process, and the time they must wait, t_{wait}.

$$\begin{aligned} E_{wait} =& 2 \times (1 - \int_0^{t_{wait}} f(t)dt)^2 \times E(\sigma_{wait}, t_{wait}) \\ &+ \int_0^{t_{wait}} f(t)dt \times E(\sigma_{wait}, t_{wait}) \end{aligned} \tag{3.6}$$

Combining these above equations the expected energy consumed by a replica-pair while waiting can be expressed as follows:

$$E_{wait_{rep}} = p_{fail} \times E_{wait} \tag{3.7}$$

This value can then be added to $E_{soc_{rep}}$ to account for the total energy consumed by a replica-pair, both during the execution phase and the waiting phase.

3.3.4 System Level Energy Consumption

The model accounts for the energy consumed by each individual socket or replica-pair. The energy consumed by the entire system is the summation of the energy consumed by all sockets. Given a system of N sockets, the energy consumed in the non-replication is:

$$E_{sys} = E_{soc} * N \tag{3.8}$$

The energy model accounts for the energy consumed by the main and the replica, the replica-pair. The number of replica-pairs is half the total number of sockets N, therefore the energy consumed by a system using replication is:

$$E_{sys} = E_{soc_{rep}} * \frac{N}{2} \tag{3.9}$$

3.3.5 Checkpointing Energy Model

The previous sections computed the expected energy consumption for both a single socket or a replica-pair. By considering only the energy consumed by a single socket, we can determine the energy consumed if coordinated checkpointing is used by those sockets. To this end, we consider the execution time required if coordinated checkpointing is used and the energy consumed by the sockets performing the execution.

Coordinated checkpointing periodically pauses tasks and writes a checkpoint to stable storage. In the coordinated case, if any one socket fails this checkpoint is read into memory and used to restart execution. Daly [23] computes the expected total wall clock time, t_w, given the original total solve time (T_s), a system MTBF (M_{sys}), checkpoint interval (τ), checkpoint time (δ), and recovery time (R). System MTBF is dependent upon the number of sockets and the socket MTBF, M_{soc}, this assumes that socket failures are independent events. The clock time, t_w, can be expressed as:

$$T_w = M_{sys} e^{R/M_{sys}} (e^{(\tau+\delta)/M_{sys}} - 1)(\frac{T_s}{\tau} - \frac{\delta}{\tau+\delta}) \tag{3.10}$$

Note that, the system MTBF is dependent upon the number of sockets and the individual socket MTBF. In the derivation of t_w, it is also assumed that failure are independent events.

Based on the above, we can express the estimated energy required for a single socket using checkpoint and restart (CPR), E_{cpr}, as follows:

$$E_{cpr} = E_{soc}(\sigma_{max}, [0, T_w]) \tag{3.11}$$

At any given time all processes are either working, writing a checkpoint, or restoring from a checkpoint; therefore, we assume all sockets are always executing at σ_{max}.

In our analysis and simulations, we use the time-to-solution optimized checkpointing interval, τ_{opt}, defined in Equation 3.12 [23]. The optimal checkpointing interval depends upon the time to take a checkpoint, δ, and the system MTBF, M_{sys}.

$$\tau_{opt} = \begin{cases} \sqrt{2\delta M_{sys}}[1 + \frac{1}{3}\frac{\delta}{2M_{sys}}^{\frac{1}{2}} + \frac{1}{9}\frac{\delta}{2M_{sys}}] - \delta & \text{for } \delta < 2M_{sys} \\ M_{sys} & \text{for } \delta \geq 2M_{sys} \end{cases} \tag{3.12}$$

Note that regardless of the level of reliability replication provides, the scheme does not fully take into consideration the case where both the main and replica nodes fail. To address this shortcoming, checkpointing is used in conjunction with process replication. Furthermore, the derivation of the energy consumed by checkpointing only differs from the one used in shadow replication in that it accounts for the fact that replication reduces the overall system MTBF. This is due to the fact that a replica-pair has a smaller failure rate than a single socket, resulting in a lower overall system failure rate. However, when using process replication, the time optimal checkpointing interval, τ_{opt} will typically be greater than the time of the application execution because of the low system failure rate, depicted in table 4.

Node MTBF (year)	τ_{opt} without replication	τ_{opt} with replication
1	87.52	546210.54
5	207.61	2.73×10^6
10	297.66	5.46×10^6
15	366.77	8.19×10^6
25	476.38	13.65×10^6

Table 4: Comparing time optimal system checkpoint intervals (in minutes) when using process replication. Checkpoint time, δ, is 15 minutes and the system contains $100,000$ nodes.

3.3.6 Energy Optimized Execution Speed Derivation

This section will be concerned with finding execution speeds that minimize the energy consumption of the application execution. The shadow replication model enables multiple points of optimization, as enumerated in table 2. We will explore these optimizations in the context of exascale systems and detail the feasible optimizations and their constraints. First looking at optimizing a single replica-pair then expanding our optimizations to include the optimization over the entire system.

3.3.6.1 Optimization Constraints To produce energy optimized execution speeds, the energy consumption equation (found in Equation 3.2) is used an objective function along with several constraints. When performing an offline optimization of two or more execution speeds, a non-linear optimization technique is used; however, when optimizing one execution speed a closed form solution is derived. Regardless of how many execution speeds are being optimized, our constraints ensure that the optimization will always produce feasible energy optimized execution speeds, ensuring that the replica-pair will always complete the task by the targeted response time. This assumes that the targeted responsible is feasible given the amount of work and available execution speeds.

The first constraint we apply is that execution speeds cannot exceed the maximum execution speed, σ_{max}. Similarly, we ensure that the execution speeds are greater than or equal to zero. Throughout our analysis we assume that execution speeds are normalized

such that $\sigma_{max} = 1$.

The next series of constraints ensure that the work is completed by the targeted response time, t_{resp}. The first constraint is to ensure that the completion time of the main process is less than the targeted response time.

$$t_c \leq t_{resp} \tag{3.13}$$

Then, ensure the execution speed of the main is able to complete all the work in the available time.

$$\sigma_m t_c \geq W_{task} \tag{3.14}$$

One of the most important constraints is that if the main process fails, then the shadow process must be able to complete the given work, W_{task}, by the targeted response time, t_{resp}. This is known as the minimum "work constraint" and is represented by the following inequality.

$$t_c * \sigma_b + (t_{resp} - t_c) * \sigma_{max} \geq W_{task} \tag{3.15}$$

This constraint states that the speed of the shadow before failure must be fast enough that if a failure occurs, the shadow can complete the remaining work by executing no faster than σ_{max}.

Based upon the above, the optimization problem can be defined as the following:

$$\begin{aligned} &\text{minimize } E_{sys}(\sigma_m, \sigma_b, \sigma_a, \lambda, W_{task}, \sigma_{max}, p_{fail}, \sigma_{wait}) \\ &\text{subject to } t_c \leq t_{resp} \\ &\qquad \sigma_m t_c \geq W_{task} \\ &\qquad t_c * \sigma_b + (t_{resp} - t_c) * \sigma_{max} \geq W_{task} \\ &\qquad \sigma_m \geq 0, \sigma_b \geq 0, \sigma_a \geq 0 \\ &\qquad \sigma_m \leq \sigma_{max}, \sigma_b \leq \sigma_{max}, \sigma_a \leq \sigma_{max} \end{aligned} \tag{3.16}$$

3.3.6.2 Online Optimization of Speed After Failure

Observe that the speed of the shadow after failure, σ_a, is dependent upon the the shadow speed before failure, σ_b, and the time of failure, t_f. This leaves us two options: determining the execution speed either online or offline. The advantage to an online optimization is that it knows exactly how much work is remaining and can optimize the speed of the shadow process accordingly. There are three potential choices when optimizing the speed after failure: σ_a can be energy efficient, power efficient, or minimize time to solution.

To determine the energy efficient speed of the shadow after failure is equivalent to finding the energy optimal execution speed of a single process. The energy consumption of a single process is modeled in Equation 3.1. After solving this optimization, we find that the optimal execution speed is a function of the overhead power and the maximum execution speed.

$$\sigma_{opt} = \frac{\rho^{1/3}\sigma_{max}}{2^{1/3}} \tag{3.17}$$

This equation does not account for upper and lower bounds placed on the execution speed. The lower bound is the minimum execution speed that enables the shadow to complete the remaining work by the targeted response time. The upper bound is simply the maximum possible execution speed.

The power efficient execution speed is the slowest possible speed to finish by the targeted response time, t_{resp}. Not surprisingly this is also the lower bound of the energy efficient execution speed.

$$\sigma_a = (W_{task} - \sigma_b * t_f)/(t_{resp} - t_f) \tag{3.18}$$

This equation determines the work remaining for the shadow and divides that work evenly over the time between the failure and the targeted response time.

Lastly, the shadow can execute at the speed that minimizes the time to solution for the shadow. This would be accomplished by executing at the maximum execution speed, σ_{max}. To model σ_a as one of these online optimizations we can substituting the closed form solutions above into the energy consumption model. This reduces the number of variables in our objective function, thus simplifying our optimization problem.

If the supported application has communication dependencies, then we need to consider that sockets are consuming overhead power while waiting for the shadow to complete. Even

for very low overhead power values, this would indicate that the execution speed of the shadow should be optimized after failure in order to reduce the execution time which is shown in the analysis Section 4.1.3, therefore $\sigma_a = \sigma_{max}$.

3.3.7 Energy Optimized Execution Speeds for HPC

We assume that the execution speed of the shadow after failure will be determined online at the time of failure. This leaves us with two execution speeds to optimize: the speed of the main and the speed of the shadow before failure. One of the primary goals of high performance computing is to achieve maximum possible system throughput. Thus when we apply shadow replication to this environment. we assume that the execution speed of the main process should be the maximum possible execution speed, $\sigma_m = \sigma_{max}$. If no failure occurs, the task will be completed as soon as possible, in what is known as the minimum response time. We explore relaxing this constraint in section 4.1.2, where we show that the optimization framework often chooses to set σ_m to the maximum execution speed when applied to exascale class systems.

Therefore, for HPC systems, shadow replication has one speed to optimize, the speed of the shadow before failure, σ_b. Using traditional optimization techniques, we take the derivative, set the result to zero and solve for σ_b producing a closed form solution. In addition to providing a model for shadow replication this can be used to represent traditional replication. Traditional replication would be represented by letting $\sigma_m = \sigma_b = \sigma_a = \sigma_{max}$, traditional replication always executes both the main and the replica at the maximum execution speed.

3.4 CONCLUSION OF THIS CHAPTER

This chapter defined the concept of *shadow replication*; developing the computational model and then further defining the execution model. Then explored possible strategies for optimizing the execution model, along with a summary of the supported application characteristics. Using this execution model and potential search space an analytical model is developed to

express the expected energy consumption of shadow replication, traditional replication and coordinated checkpointing. Lastly, defining the optimization problem used to find the energy-optimal execution speeds for shadow replication. The next chapter will use these analytical models to explore the potential energy and power savings offered by shadow replication.

4.0 ENERGY AND POWER ANALYSIS

The previous chapter introduced the computational and execution model of *shadow replication* and developed the analytical framework for determining the expected energy consumption of *shadow replication* along with other fault tolerance methods. This chapter discusses an energy-based comparative analysis of these fault tolerance methods. The primary goal of this analysis is to provide a strong justification for implementing shadow replication in exascale environments by demonstrating significant energy and power savings. The secondary goal is to explore the design space of shadow replication to better understand implementation concerns and limitations by exploring sensitivity to system characteristics.

The first part of this chapter is an analysis of the potential energy savings over traditional replication. This analysis provides insights into how *shadow replication* behaves and how optimizations should be applied. Additionally, the energy analysis will confirm that in exascale HPC environments the energy optimal execution speeds for the main, σ_m, and shadow after failure, σ_a, would be the maximum execution speed. The second part of this chapter focuses on the application of *shadow replication* to exascale-class power-limited environments. That analysis will compare *shadow replication* to traditional replication, stretched replication, and coordinated checkpointing demonstrating that shadow replication could potentially provide remarkable energy savings in those environments. The last part of this chapter summarizes the major findings of the shadow replication analysis.

4.1 ENERGY ANALYSIS

Previous studies have shown that replication will become more efficient than coordinated checkpointing in exascale systems [13, 38]. Based upon the outcomes of those studies, the first performance study carried out in this chapter limits the analysis to the comparison of shadow replication to traditional replication. The focus of the analysis will be on energy savings in exascale computing environments with different computational characteristics, including the number of nodes, laxity factors, MTBF values, and power constraints.

Intuitively, one may assume that shadow replication is bound to achieve higher levels of energy savings than traditional replication. Although that is likely to be true in communication and synchronization-independent environments, the interplay between power savings and time to solution, coupled with the impact of idle waiting caused by communication dependency, puts such an assumption into question and requires further investigation. Additionally, there is a known fundamental tradeoff between power savings and time to solution.

To carry out the energy savings study, the energy optimization model is used to derive the optimum execution speeds for the main and its associated shadow. This analysis constrains the exploration to the computational aspects of HPC applications; namely, communication dependency. As discussed in the analytical framework, the need for communication synchronization may give rise to idle waiting as processes complete their execution. Based on the optimal execution speeds, the total system energy consumption is computed for both shadow replication and traditional replication. Using these values this section presents the potential energy savings by showing the percentage of energy savings.

In summary, throughout this analysis, it is assumed that the application exhibits full communication dependency. As a result, σ_m and σ_a are set to σ_{max}, while σ_b, the only remaining optimization variable, is the derived solution of the energy-optimization problem. Unless otherwise noted, throughout this analysis σ_b is the energy optimized execution speed, as described in detail in Section 3.3.6. Furthermore, without loss of generality, all execution speeds are normalized such that $\sigma_{max} = 1$. This eliminates the dependency on specific hardware characteristics of the executing sockets.

4.1.1 Comparison to Traditional Replication

Our analysis begins by looking at the effect node MTBF has upon potential energy savings. To begin our analysis we study the impact of the node MTBF upon the potential energy savings. In Figure 10, we observe that shadow replication exhibits potential energy savings over traditional replication, 4-24% depending on node failure rate and the value of the laxity factor. Additionally, when the socket MTBF reaches 15 years, the amount of additional energy savings achieved by shadow replication no longer changes significantly. This is due to the fact that in systems where node failure is an extremely rare event, shadow replication is optimized to execute at the slowest possible speed in order to conserve power. Two factors prevent the speed of the execution to go below a threshold, determined by the minimum work constraint and the overhead power. As a result, the maximum possible energy savings is bound to 25%.

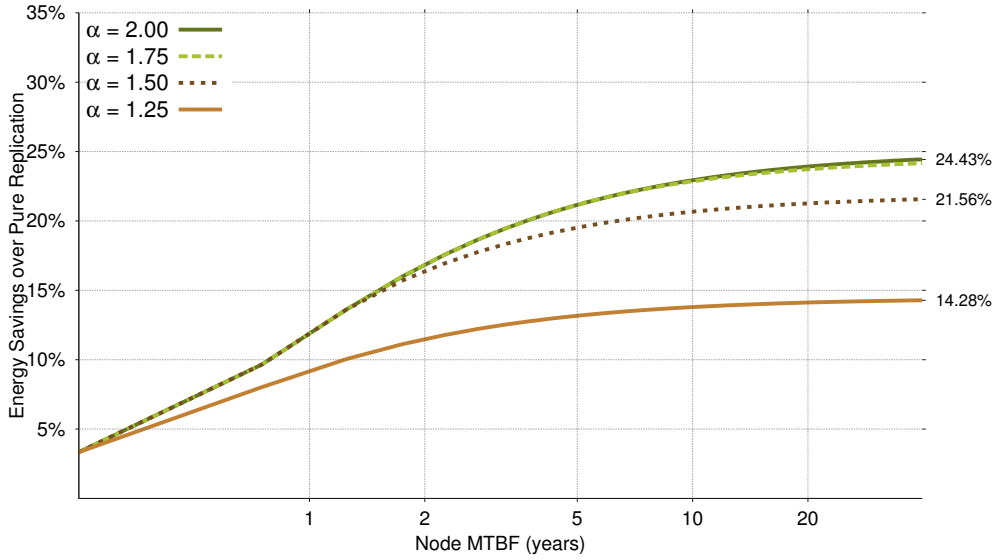


Figure 10: Energy savings of shadow replication for various alpha values, varying mtbf. $100,000$ nodes, Static power 50%.

The next analysis studies the potential energy savings as the system size increases. Figure 11 shows, that the number of sockets effects both replication techniques in similar ways. While the energy consumed will increase, it increases by the same amount regardless of the

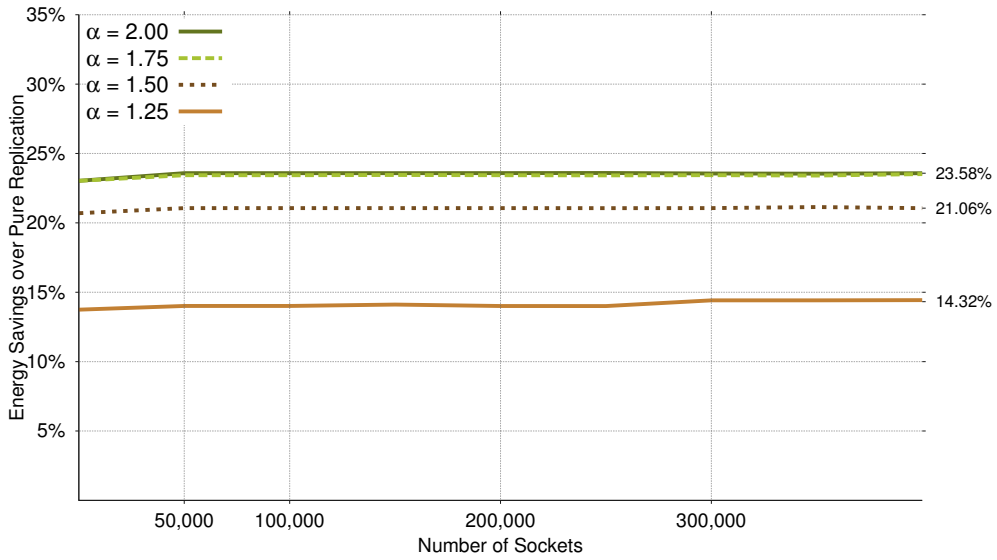


Figure 11: Energy savings of shadow replication for various alpha values, varying number of sockets. 15 year socket MTBF, Static power 50%.

replication technique, making the energy savings fairly constant. Because we are computing the expected energy consumption, the model captures only the average case behavior. In the average case, the node MTBF and the number of nodes will have almost the same effect on both replication techniques. The only difference is in the energy consumed by the processes having to wait for the slowest node to complete. As the MTBF increases, this waiting time has little effect upon the overall energy consumption, making it behave very similarly to traditional replication. In the simulation work we explore more complex interactions between the processes of shadow replication which the analytical model cannot express.

Two variables that have a significant effect upon the energy savings are the amount of laxity, represented by α, and the percent of overhead power. The laxity variable, α, is the lever that allows us to trade off between time and hardware redundancy. If there is no laxity, ($\alpha = 1.0$) shadow replication requires full replication, which in turn requires full hardware redundancy and saves no energy. As we increase the laxity, shadow replication can trade off hardware for time, and also conserve energy along the way. Observe in Figure 12 that once the laxity become twice the minimum response time, shadow replication can no longer

save additional energy, because once there is twice as long to complete one would always choose to perform re-execution instead of any form of replication. We detect this in our optimization because the energy optimized execution speed of the shadow before failure is calculated as zero.

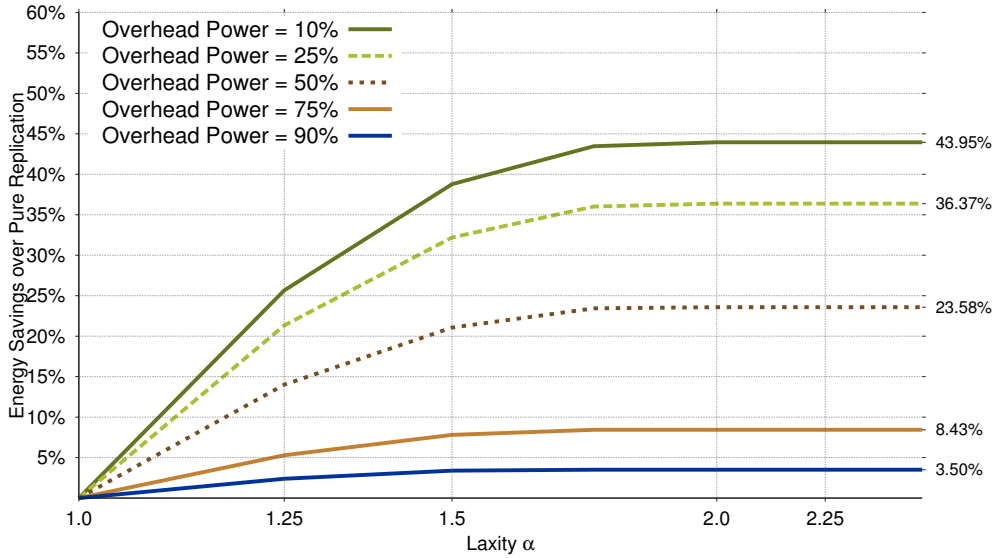


Figure 12: Energy savings of shadow computing for various alpha values, varying static power. 15 year socket MTBF, 100,000 nodes.

Thus far we have considered the potential energy savings assuming that the overhead power per socket was 50% of the power consumed when the CPU was executing at maximum execution, as detailed in Section 3.3.1. By considering the overhead power, we effectively limit the savings achievable by shadow replication. In our model, even if the processor is executing at zero the socket will continue to consume fifty percent of that consumed by a socket running at maximum speed. Fifty percent overhead has been chosen because our experiments[77] and others [43, 64] have shown that current architectures have an overhead of 40-60%. Its expected that these overhead will remain consistent in the future.

Observe in Figure 12 that as the overhead power is increased, the amount of energy savings decreases. When the overhead is at 10% we see a that the potential energy savings can be as high as 43.95%, although at the more realistic 50% the potential savings reduces to 23.58%, as we have previously seen.

4.1.2 Optimizing Execution Speed of the Main

As discussed in the previous chapter, specific knowledge of the HPC environment and the exascale application it supports may further constrain the energy optimization model used to derive the optimal speeds of the main and its associated shadow. Furthermore, it was argued that when using shadow replication to achieve fault-tolerance in HPC environments, the execution speed of the main, σ_m, must be set to σ_{max}. This assumption is justified by the critical need to achieve high throughput in HPC environments. It is further motivated by the high infrastructure cost, which makes idling computing and communication hardware financially prohibitive during the execution of an application. This section further considers the decision to set σ_m to σ_{max}, presenting a relaxed formulation of the energy optimization model defining σ_m as an optimization variable. The optimized solution shows that for most of the exascale design space, the optimization model derived value of σ_m is σ_{max}.

The hypothesis was that by relaxing the model to allow σ_m to be a optimization parameter the potential energy savings of shadow replication would increase. Using Mathmatica's non-linear optimization methods the model was relaxed and then used to find optimal execution speeds for both the speed of the main, σ_m, and the speed shadow before failure, $sigma_b$. Calculating the energy savings using these optimal values in HPC environments showed little or no difference in energy savings. Upon further review we found that if the overhead power percentage was 50% or higher the optimization found σ_m to also be σ_{max}, as depicted in Figure 13. Because it is expected that overhead power will be between 40-70% in exascale environments shadow replication analysis was unable to achieve power savings beyond those already observed.

4.1.3 Optimizing Execution Speed of the Shadow After Failure

Exascale applications are massively parallel and tightly coupled, and they exhibit full communication dependency among their components. Consequently, processes executing in parallel can only move forward if their communication requirements have been satisfied. A direct consequence of this requirement is that processes must wait for the slowest process to complete before they can resume execution. In Chapter 3, we show that the general formulation

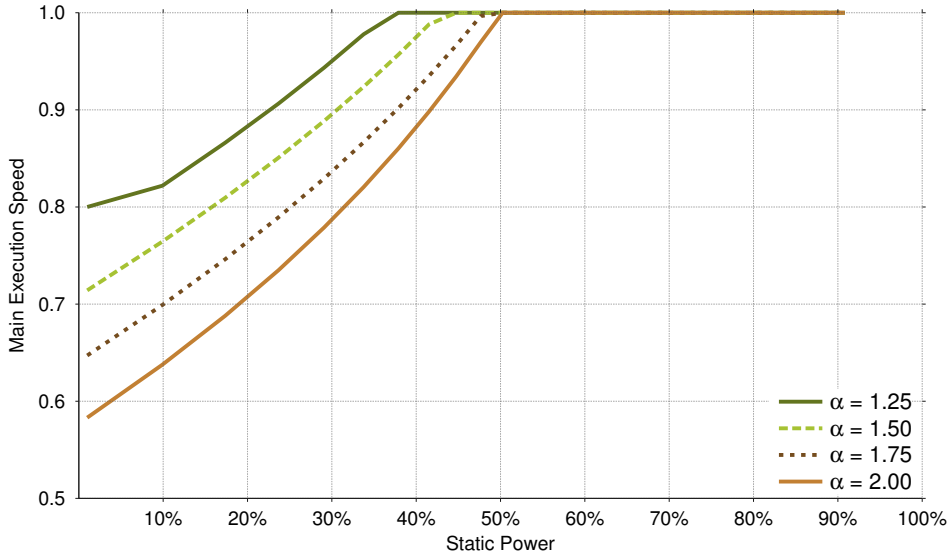


Figure 13: Energy Optimal Main Execution Speed, σ_m, vary static power. 5 year socket MTBF, 100,000 nodes.

of shadow replication energy optimization model leads to an online optimized value, σ_a, of the shadow speed after failure. Such a formulation, however, ignores the communication dependency among the application's processes. The need to reduce the time that processes may idly wait while waiting for the last shadow of the failed main process to complete requires that the value of the shadow execution speed after failure, σ_a, be σ_{max}.

To confirm this intuition the model is relaxed to derive the energy optimized execution speed of the shadow after failure, σ_a. This is performed using Mathmatica's non-linear optimization to determine the offline energy optimal execution speed of the shadow after failure for different communication dependency types. Figure 14 shows the results of this optimization for different application types. If there is any communication dependency the energy optimal speed after failure becomes the maximum execution speed once static power is over 40%. There is one case removed from Figure 14, which is when static power is zero the execution speed of the shadow after failure does not effect the waiting processes, resulting in a slower execution speed than is plotted.

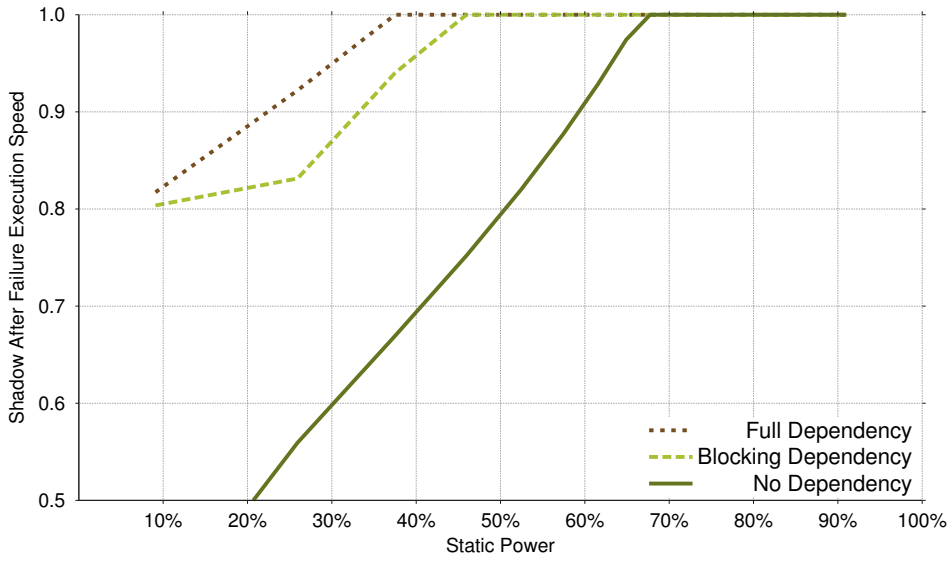


Figure 14: Offline energy optimal shadow execution speed after failure, σ_a, for different application types. 5 year socket MTBF, 100,000 nodes.

4.1.4 Sensitivity to Buffer Size

Another constraint we have not considered is the buffer size available to hold messages destined for slower running shadow process. The shadow replication library will be responsible for holding messages destined for the shadow process until the shadow is ready to process them and these buffers will undoubtedly be limited. One method for working with limited buffers is to set a lower bound on the speed of the shadow prior to failure. This bound forces the shadow to consume a given rate of messages, while keeping the buffers from overflowing. Using this method introduces another constraint on our optimization problem.

Let r represent the rate at which the main process would consume messages. This rate would be defined as bytes per cycle and will be dependent upon the speed of the main process, σ_m. The buffer size, b, is defined as the number of bytes available for buffer for an individual node. We can then express the speed before failure, σ_b, in terms of the speed of the main process, σ_m, by defining a speed factor θ.

$$\sigma_b = \theta\sigma_m \tag{4.1}$$

Application	Msg Log Growth Rate (Bytes/sec.)
CTH	5.42×10^7
miniFE	2.58×10^6
LAMMPS	2.16×10^6
AMG	9.66×10^5
HPCCG	7.43×10^5

Table 5: Maximum per-process message log growth rates, including both point-to-point and collective operations for a number of production HPC workloads.

If we assume the rate of message consumption is linear, with respect to the speed, we can define the buffer constraint as the following.

$$(W_i/\sigma_m) * (1 - \theta) * r \leq b \tag{4.2}$$

This expresses that the rate of message consumption must be such that the size of the messages present in the buffer should not exceed b over the entire execution time, W_i/σ_m. Solving for θ,

$$\theta \geq 1 - \frac{\sigma_m * b}{W_i * r} \tag{4.3}$$

Lastly, one can obtain the constraint on σ_b by combining the equation 4.1 and equation 4.3.

$$\sigma_b \geq (1 - \frac{\sigma_m * b}{W_i * r})\sigma_m \tag{4.4}$$

To evaluate these buffer size constraints we ran a number of representative HPC workloads and used the rMPI to collect the mean message logging growth rate of the application. These workloads include the production applications CTH [25], a shock physics code, LAMMPS [87], the molecular dynamic code, and the algebraic multi-grid solver AMG [66]. We also include results from two of the mini-applications from Sandia's mantevo suite [97]: HPCCG (a conjugate gradient solver) and miniFE (an implicit finite element method). These applications represent a range of computational techniques, are frequently run at very large scales on leadership class systems, and represent key simulation workloads for the U. S. Department of Energy. Using the modified rMPI replication library, we measured the maximum per-process message log growth rate for each application, shown in Table 5. From this table,

we see message log growth rates can vary dramatically. For example, CTH, which does a good deal of bulk data transfer, has the largest growth rate of the applications tested, a measured 54MB/sec, while AMG, which does much less communication, has a log growth rate of nearly 1MB/sec.

To analyze the impact buffer sizes have upon the potential energy savings of shadow replication, one can look at the effect the constraint has upon the energy optimal execution speed of the shadow. To do this one applies the buffer constraint defined in Equation 4.4 to the optimization problem. Figure 15 shows that when the constraint is applied, the speed of the shadow before failure is forced to increase as the work size increases. The constraint forces the execution speed to exceed the energy optimal execution speed, ultimately reaching σ_{max} and causing shadow replication to mimic traditional replication. Once this occurs, there is no potential energy savings offered by shadow replication. Applications with high message rates, such as CTH, a faster execution speed than those applications with lower message rates such as LAMMPS.

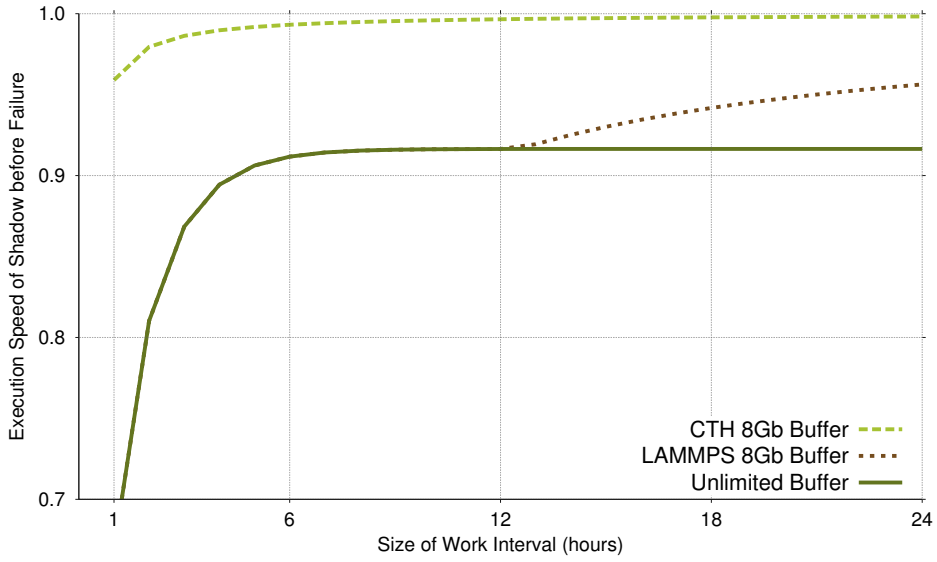


Figure 15: Effect buffer constrains have upon the speed of the shadow before failure. 5 year socket MTBF, 100,000 nodes.

4.2 POWER ANALYSIS

The previous section analyzed the expected energy savings when given a fixed number of nodes. However, it is expected that exascale-class machines will be capable of consuming more power than what is set by the DoE's target system power limit of 20MW [2]. For example a system might have 150,000 sockets, each consuming 200 watts of power at full speed. Therefore, if all sockets were operating at full power, we would be consuming 30MW. To stay under the 20MW limit, 50,000 of these sockets would need to be powered off, or the power consumption of some or all of the cores would need to be reduced in order to stay within budget. While this may seem inefficient, as more hardware is available than can be supported by the power infrastructure, not all applications will be capable of fully utilizing the system.

Given a power limit and socket power consumption there will be a fixed number of sockets available at any one time. Coordinated checkpoint/restart would use all of the available sockets to perform work and in the event of failure would roll back all sockets, therefore staying under the power limit by restricting the number of sockets used. Traditional replication would take half of the available sockets and use them as replicas, which has been shown to increase system efficiency in exascale-class machines. In contrast, shadow replication has the ability to use additional sockets because the replica sockets are consuming less power by running at a reduced speed. This has the added benefit of allowing additional sockets to work as main processes while still providing system resilience such as that seen in traditional replication. In the event of failure, there is the potential delay in the time to solution because of the replica's slower execution speed. However, because of the ability to use additional sockets, we show that the expected time to solution is actually faster than both checkpoint/restart and traditional replication methods when accounting for the power limit.

Our analysis finds several system parameters to be important in determining which fault tolerant method is most efficient.

- I/O Bandwidth - This dictates how long it will take to write or recover a checkpoint.
- System Size - The number of total sockets.

- Socket MTBF - Reliability of a single socket in the computing system.
- Overhead Power - The overhead power consumed by the socket, as described in Section 3.3.1.

When comparing fault tolerant methods, we calculate the energy consumption and time using the power, failure and energy models described in the previous section.

4.2.1 Stretched Replication

In this analysis we also consider a simple power-aware replication technique called stretched replication. Stretched replication works on the assumption that performing work slowly can save energy. Stretched replication is a naïve approach which slows down the execution of all processes to the slowest possible speed while maintaining the applications targeted response time.

Previous analysis dismissed this approach because overhead power dominated any potential savings. Specifically, in direct energy analysis stretched replication performed more poorly than traditional replication when overhead power exceeded 50% and laxity values of 1.25. Because of this we choose not to present stretched replication as a solution in the previous analysis. However, stretched replication enables the execution of more nodes than traditional replication, which is of benefit in power-limited environments. Accordingly, stretched replication has been included in this sections analysis.

4.2.2 Scaling and Failure Rates

Break-even values are calculated by computing the energy and time required for coordinated checkpointing and comparing that to the energy and time required for the replication technique. The break-even point is when those values match, We compare fault tolerance efficiencies by identifying the break-even point at which the replication technique is equivalent to that provided by coordinated checkpointing. We use two different break-even metrics: the expected energy consumed, and the expected time to solution. While these are related to one another; because energy is a function of time, due to overhead power they are not

equivalent. All of the area above the break-even curve is where the replication technique is more efficient than coordinated checkpointing.

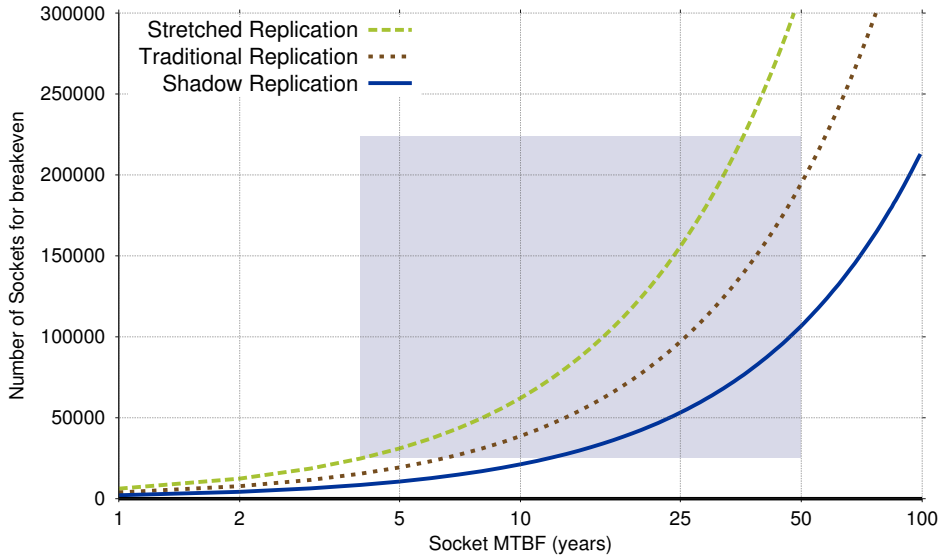


Figure 16: Breakeven points for energy given a fixed checkpoint time of 15 minutes and a system overhead power of 60% with barrier communication dependency.

Figure 16 shows the energy break-even point varying system size and socket MTBF using a fixed checkpoint time of 15 minutes. These results show that shadow replication can provide a significant energy savings over traditional replication. For example, when socket MTBF is 25 years, traditional replication is viable at $96,700$ sockets whereas shadow replication is more efficient at $53,100$. This represents a 46% energy efficiency gain. Unfortunately, stretched replication is less energy efficient because of the increased time to solution and the presence of overhead power.

Shadow replication achieves this energy savings by slowing down the replicas. This raises the question of how this will affect the expected time to solution. Figure 17 plots the time to solution break-even point, and shows that even though shadow replication slows the replicas, the expected time to solution is actually shorter than that provided by traditional replication. For example, when socket MTBF is 25 years, traditional replication is viable at $97,600$ sockets, whereas shadow replication is more efficient at just $52,700$ sockets, representing a 46% improvement in expected time to solution.

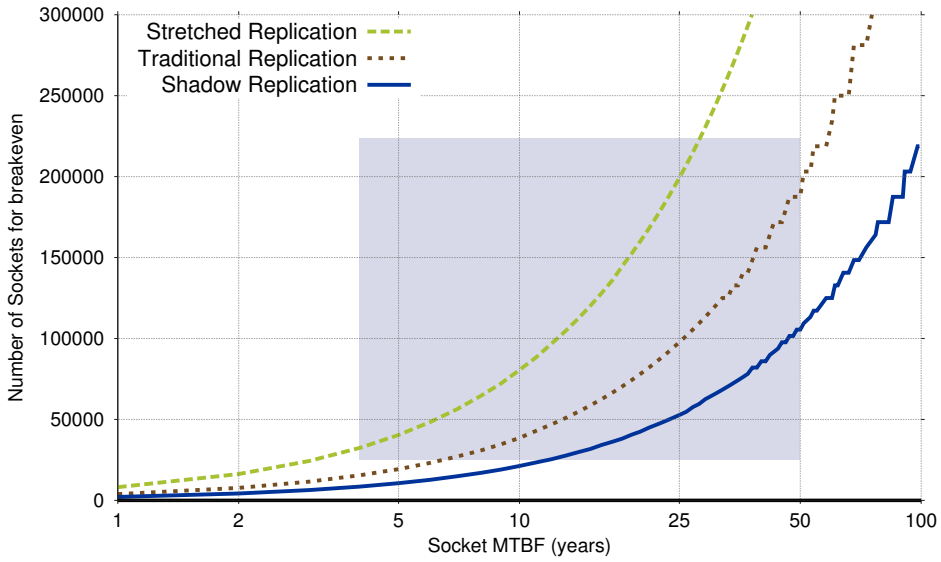


Figure 17: Breakeven points for time given a fixed checkpoint time of 15 minutes and a system overhead power of 60% with barrier communication dependency.

The improvement in time to solution arises because shadow replication can utilize additional sockets while consuming the same power, because the replicas are consuming less power. This is illustrated in Table 6 which shows the active socket counts allowable given a 20 megawatt fixed power budget. Both stretched and shadow replication have the ability to use additional sockets because they reduce the power consumed by the individual sockets. Stretched replication reduces the power consumed by all processes equally whereas shadow replication only reduces the power consumed by the replica sockets. This is the reason that the expected time to completion of shadow replication outperforms traditional replication. Stretched replication is able to add additional nodes but because it also reduces the processor speed of the main processes, the time to solution is higher than both traditional and shadow replication.

In pure replication, the total amount of work remains constant but the the number of sockets is half of that available to coordinated checkpointing. Our model assumes a strongly scaled application, which is a fair comparison because each socket would have less work to accomplish than in coordinated checkpointing. Thus, in a failure free case it would be

Overhead %	Method	# Sockets	# Main Sockets
60%	Checkpointing	100,000	100,000
60%	Traditional Replication	100,000	50,000
60%	Stretched $\alpha = 2.0$	153,846	76,923
60%	Shadow $\alpha = 2.0$	124,998	62,499
80%	Checkpointing	100,000	100,000
80%	Traditional Replication	100,000	50,000
80%	Stretched $\alpha = 2.0$	120,230	60,115
80%	Shadow $\alpha = 2.0$	110,636	55,318

Table 6: Available sockets assuming a 20 mega-watt power limit and 200W per socket.

faster than replication techniques. However, because with replication there are two sockets instead of one, the MTBF for the pair is greater than that provided in the single-socket case. The change in MTBF is what allows replication to outperform coordinated checkpointing on a large scale. In shadow replication, instead of assuming half of the original sockets are replicas, we calculate the energy optimal σ_b for $\alpha = 2.0$. We then "add" additional sockets, all the while remaining under the original power limit, but continuing to use half the sockets as replicas. Stretched replication is similar to shadow replication, but both the replica and main use less power.

We conclude that shadow replication is both more energy efficient and produces solutions faster than traditional replication in power-limited systems. This is true for the majority of the exascale design space, illustrated by the region in the grey box in Figure 16 and 17. We assumed a fixed checkpoint time of 15 minutes and a overhead power of 60% which are reasonable system parameters given expected exascale I/O bandwidth and increased system efficiencies [37]. In the next sections we further relax these assumptions and study the models sensitivity to these parameters.

4.2.3 Scaling at Different Checkpoint I/O Rates

The checkpoint write times have a significant effect on the efficiency of coordinated checkpointing. These times are directly related to the available I/O bandwidth, as modeled in [81]. Figure 18 uses these models to determine the energy break-even points for I/O bandwidth

rates from 500GB/s to 50TB/s, representing a wide range of values for an exascale-class machine. For space reasons, we only show results for shadow replication, though other replication techniques follow a pattern similar to that in Figures 16 and 17. Shadow replication is viable for all but very extreme levels of I/O bandwidth.

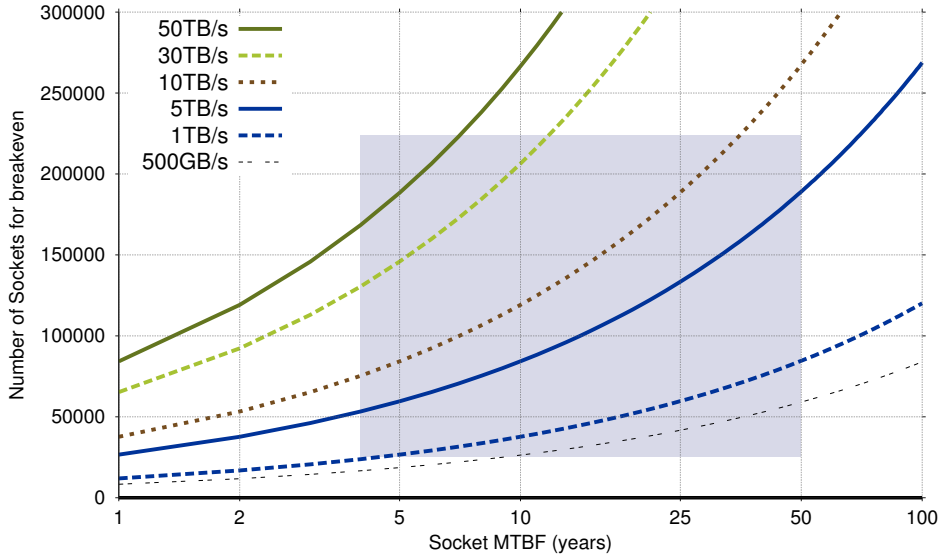


Figure 18: Shadow replication energy breakeven for different I/O bandwidths. Assumes 16Gb per socket with barrier communication dependency.

4.2.4 Scaling at Different Overhead Power

Table 6 illustrates that the number of available sockets decreases as the percentage of overhead power increases. Shadow replication can only reduce dynamic power consumption, leaving it with less power headroom to improve efficiency. This means it can take advantage of fewer main sockets as the available power headroom decreases. Figure 19 shows the effect overhead power has upon the energy break-even point. As the power overhead increases, the potential energy savings also decreases, moving the break-even point further out into the exascale domain. The conclusion is that overhead power does have an effect upon shadow replication, but even if the overhead is 100% it will be no worse than traditional replication. It is expected that future hardware will reduce this overhead, making shadow replication more efficient.

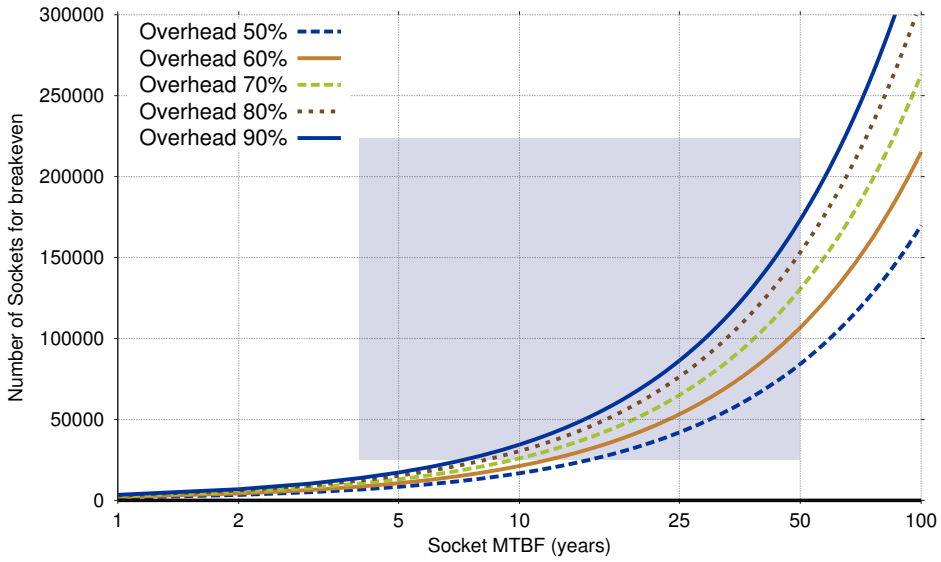


Figure 19: Shadow replication energy breakeven for various overhead power percentages. Checkpoint time of 15 minutes with barrier communication dependency.

4.3 CONCLUSION OF ANALYSIS

This utilizes an analytical framework to demonstrate that shadow replication is 25% more energy efficient than traditional replication for system parameters expected in exascale-class systems. We also demonstrate that for strong scaling applications in power-limited environments shadow replication can be over 40% more energy and time efficient than traditional replication. Moreover, this chapter shows that replication techniques are more efficient than checkpoint/restart techniques for much of the exascale design space and that shadow replication is the most efficient replication technique presented. Ultimately, we have shown that shadow replication has potential to save significant energy when compared to existing fault tolerant techniques.

These promising results provide a basis for continuing to explore shadow replication as a fault tolerant model in exascale-class systems. Additionally, this chapter identifies several challenges that will need to be addressed during the implementation and areas which will require further exploration to fully understand the potential energy savings. Due to the

reduced speed of the replicas one concern is the growth of the message queues between replica and their leaders, which occupy memory on the replica nodes. This chapter shows that the rate of this queue growth is highly dependent upon an application message rate. A solution presented is to constrain the execution speeds of the processes such that messages are consumed before they overflow the buffer. There are other potential options for solving this problem explored in future work, including a the concept of "jumping shadows" and using properties of the applications such as send-determinism [45]. Lastly, this chapter identifies that inner-process communication dependency is a critical factor in the amount of achievable energy savings. The analytical framework accounts for this dependency but does not fully model all the intricacies of process communication; specifically failures of nodes while waiting. To explore this limitation the next chapter will build a simulator that is capable of emulating these communication dependencies.

5.0 SIMULATION AND IMPLEMENTATION

The analytical model has shown that shadow replication is a fault tolerant method that reduces energy consumption while also increasing time to solution in highly scalable applications. However, the analytical framework was not designed to fully capture the complex communication between processes, as would be present in tightly-coupled HPC applications. Therefore, in this chapter we will develop a simulator, HPCSim, that seeks to more accurately depict the computing environment found in HPC and to evaluate different fault tolerant techniques including checkpointing, replication and shadow replication.

5.1 SIMULATOR OVERVIEW

The goal of the simulator is to emulate multiple nodes, each executing part of a tightly coupled application. Failures can occur at each individual node and are assumed to be independent from one another. Each node consumes energy while executing but is adjustable by reducing the execution speed of the node. We define clusters of nodes which work in parallel to solve a problem of some fixed size. Using these primitives we then simulate checkpointing by periodically pausing execution and saving all the nodes state. To simulate replication we combine two or more nodes together and compute their energy consumption and failure rates accordingly. Similarly, shadow replication is simulated as a replica-pair, however the shadow process is executed at a speed independent of the execution speed of the main. This framework allows us to measure the energy consumption and time to solution for different applications while simulating an exascale environment with its potential failures.

HPC applications are tightly-coupled, therefore all nodes in a cluster must successfully

execute in unison without failure to complete the job. This is because HPC applications all work on part of the same problem and constantly communicate with one another in order to solve the problem. If a failure occurs at a node then all the work completed on all the node is lost and all the nodes must complete without failure in order to find the solution. The simulator has the ability to simulate all three process communication types: no dependency, barrier dependency and full dependency. Recall that these correspond to the level of communication dependencies between the processes. No dependency refers to application that can execute without communication between their processes. Barrier dependency means that the processes can perform their work independent of communication, but must communicate at the end of the task to produce a final result. Full communication refers to applications that require all processes to communicate with each other during the entire execution.

The simulator causes a failure at each node, independently of each other using a probability distribution function. The parameters of the distribution function are configured in the simulator to describe the behavior of each individual node. When a failure occurs the node immediately stops its work and issues an event announcing its failure, then other parts of the simulator react to this failure.

Most HPC systems deploy checkpointing to help mitigate this risk, by periodically writing all the nodes' state to stable storage in order to restart the execution from the last known failure-free state. We simulate this behavior by periodically pausing execution and waiting for the checkpoint to complete. The checkpoint interval can either be a specified amount of time or can be optimally computed using Daly's equations. The time necessary to write the checkpoint can be either a fixed amount of time or dynamically calculated based upon the system bandwidth and the size of state recorded by each node. While the checkpoint is being executed we assume the nodes continue to execute at their previous execution speed. As we show in Chapter , there is a potential to save energy during these checkpoint events by reducing the execution speed, however we do not explore this behavior in this chapter.

To simulate traditional replication and shadow replication we replace a single node with a pair of nodes. The pair of nodes have all the same functions and parameters of the individual nodes. However, the failure of one node in the pair does not cause the pair to fail, instead

the replica node simply takes over the execution. If both nodes fail, this triggers a failure in the entire cluster, resulting in all nodes re-executing from the last known checkpoint, just as it would in the single node case. In order to simulate shadow replication the replica node in the pair is executed at a slower execution speed. Accordingly, if the main node fails, the amount of work done by the pair will be "reduced" to the amount of work done by the replica process which is less than that done by the main. However, just as in traditional replication, this failure will not result in a rollback of all other nodes.

Adjusting the execution speed of a node changes both the amount of work being completed and the energy consumption of the node. The amount of work being completed is linearly reduced as a function of the execution speed. To record the amount of work completed each node maintains a log of its execution time and speed.

We also account for the overhead power consumed by the node. Similar to our analytical model we assume this is a fixed amount of overhead, proportional the energy consumed by the node when executing at full power.

5.2 IMPLEMENTATION

The simulator was implemented using SimPy, a Python based discrete event simulator library. SimPy uses Python generators to populate the discrete event queue and allows for event driven simulations using using a simple signal handles. Processes in SimPy are Python generator functions that then model machines, clusters, and checkpointing functions. SimPy also provides various types of shared resources to model limited capacity congestion points, for example CPUs and power. It also provides monitoring capabilities to aid in gathering statistics about resources and processes.

5.2.1 Architecture Model

The architecture of the system was designed as a series of interchangeable components. The core object is called `Machine` which simulates the execution of a node, tracks its energy

consumption, records the amount of work completed, and fails according to the specified distribution and parameters. When a `Machine` fails, it must decide what to do next. If a parent exists, it reports the failure to its parent, if such a parent exists. In the simplest case, the parent is a `Cluster` object which holds pointers to a series of machines. When a failure occurs, the cluster must decide what to do next. In our default implementation the cluster reports the failure to its parent if one exists. The typical parent of a cluster is another object called the `Worker`, the worker is responsible for knowing how to respond to errors, to track the amount of work completed and when to stop working. The default worker declares failure when the cluster fails without completing its work. There is also a `CheckpointWorker` which given a checkpoint interval, checkpoint write time, and recovery time emulates running a cluster and periodically checkpointing the cluster's state. When a cluster failure happens during execution, the worker waits the recovery time and restarts the cluster.

In order to simulate both traditional and shadow replication there is another machine specified called `ShadowMachine`, which operates much like a cluster of two nodes. However, when a failure occurs in one of the nodes it decides how to handle that failure based upon the simulator's configuration. If it is acting as a traditional replica machine, no changes are made and the failure event is not propagated. If the machine is acting as a shadow replica machine, and the failure occurs in the main machine, then it adjusts the execution speed of the shadow. If the shadow machine fails then no change is made. In either case the single machine failure event is not propagated beyond the `ShadowMachine`. If both machines fail in the `ShadowMachine`, the failure event is propagated to the shadow's parent.

The idea is to divide the logic of concerns among a series of objects in order allow for simple re-configuration for different simulations and to easily audit their correctness. The simulator is not optimized for memory usage but is instead optimized for flexibility and code-readability. Even so we have been able to simulate executions of up to 12 hours on clusters of 250,000 nodes using a machine with 12Gb of memory.

5.3 VALIDATION OF SIMULATOR

The validation of the simulator occurred in two phases. First, we began with known edge cases and calculated the measured metrics by hand, and then confirmed that the simulator provided those expected results. Second, we ran simulations that matched the system configuration used in the analytical model and confirmed that the output closely matches.

5.3.1 Unit Tests

To ensure individual components of the simulator were functioning correctly we compared simulation outputs to the hand-calculated expected results. This was done by writing unit tests for each component in the simulator and verified by calculating the expected output. These tests were then run periodically while development was done on the simulator to ensure that no regressions had been introduced.

5.3.2 Comparison to Analytical Model

Assuming that our models are accurate, the simulator should produce results similar to that observed in the analytical models. The simulator captures the intricacies of inner-socket communication so it is expected that the simulation will differ from the model in those cases. It is expected that no communication dependency cases should match. Figure 20 plots both the projected energy savings found by the analytical model and those found by the simulator for applications with no communication dependencies. As can be observed, the results are nearly identical.

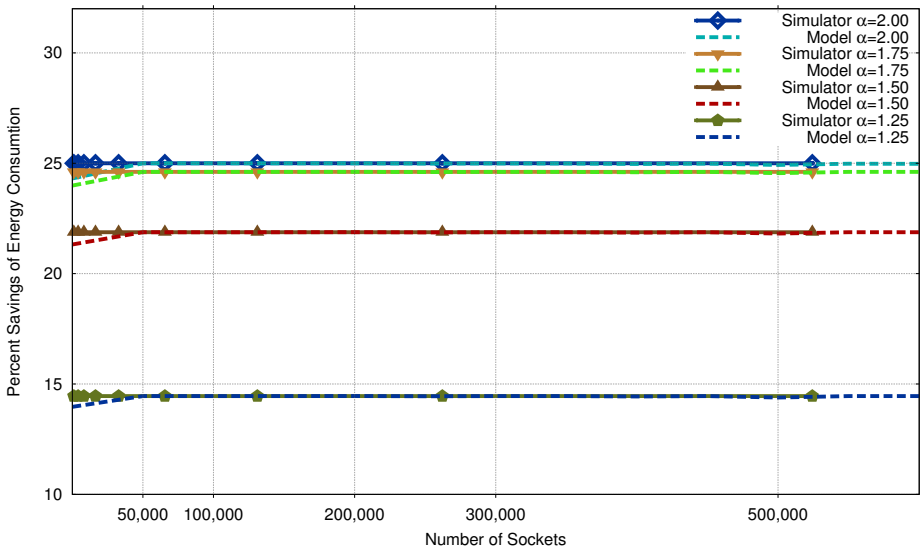


Figure 20: Simulated energy savings compared to that predicted by the model for application with no communication dependencies. MTBF=25 Years, Work = 2 hours at full speed, Socket Power Consumption=200 Watts, Static Power 50%

5.4 SIMULATION RESULTS

5.4.1 Energy Consumption of Shadow Replication

Our analytical model showed that shadow replication would be able to save 14-24% energy over traditional replication. The results obtained from the simulator did not show this potential energy savings; in fact, it showed that shadow replication increased energy consumption as the number of nodes were increased, shown in Figure 21. This can be explained because the expected energy model did not fully model the application communication; specifically, it does not capture the cascading delay effect, explained below in full but defined as the delay of one process because another process has failed, which in turn makes the delayed processes more susceptible to failure. As the number of nodes increases, the overall system failure rate increases, resulting in additional cascading delay and eroding any energy savings achievable by shadow replication.

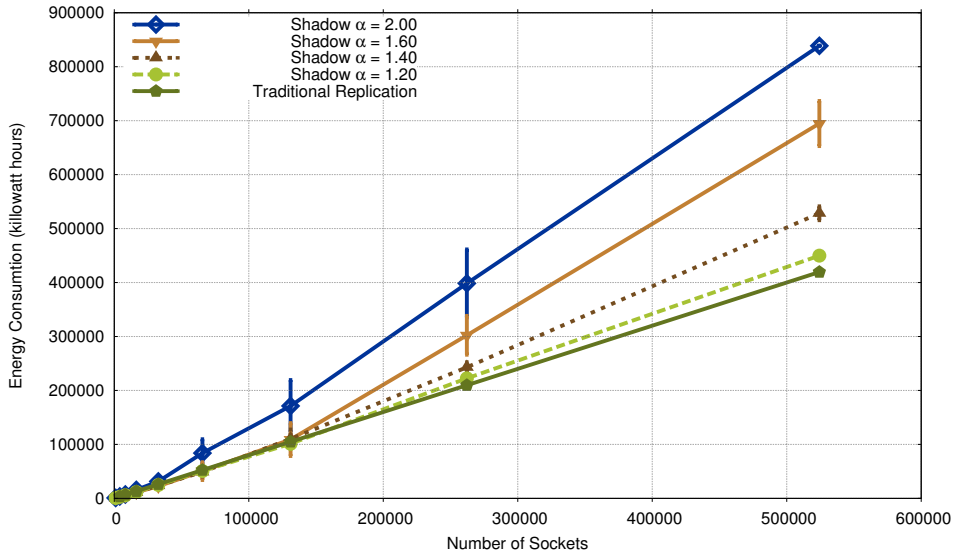


Figure 21: Simulated energy consumption, comparison between traditional replication and shadow replication. Node MTBF=25 Years, Work = 2 hours per socket at full speed, Socket Power Consumption=200 Watts, Static Power 50%

As stated above, cascading delay is the delay of one process because another process has failed, which in turn makes the delayed processes susceptible to failure. This can cause the time to solution to increase beyond the targeted response time, t_r, as illustrated in Figure 22. Cascading delays increase in frequency as the likelihood of a system failure increases, such as when the system size grows, demonstrated in Figure 21. As the system scales and failures become more common the time to solution increases beyond the available laxity, α. As depicted in the simulation data a job requiring two hours of work with laxity at $\alpha = 1.4$ should take no longer than 168 minutes. However, as the simulation data demonstrates it can increase to well beyond the targeted response time in exascale-class systems. This causes the overall energy consumption to increase and therefore negates the potential energy savings showed in our analytical model. The time to solution is ultimately bounded by the speed of the shadow before failure.

$$t_{bound} <= W_{task}/\sigma_b \tag{5.1}$$

This limitation arises due to the fact that in the worst case all tasks will be completed by the shadow process which is executing at the speed of the shadow before failure. The problem in our optimization is that as the laxity approaches twice the job size the speed of the shadow before failure approaches zero and ultimately creates a nearly unbounded delay. This is why in our simulations the smaller the laxity, the better the results.

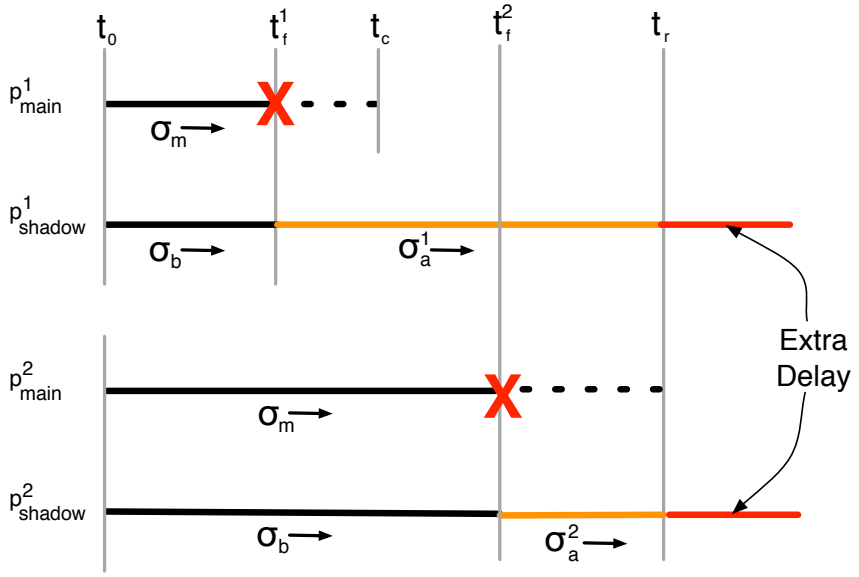


Figure 22: Illustration of cascading delay involving two shadow processes.

5.4.2 Simulation of Power-Limited Environments

In the previous simulation results we assumed that the work for each socket was fixed regardless of the number of sockets, which mimics a weak/constant-scaling application. This section assumes, a power-limited environment in which there are more computing resources than there is power to turn them on. Furthermore, we assume a perfect strong-scaling application that can utilize all available computing resources to complete a fixed amount of total work. This is the same set of assumptions made in the power analysis performed in Section 4.2.

In power-limited environments the simulations demonstrated that there is the potential to save energy over traditional replication, although the saving is dependent upon the

applications communication pattern. Figure 23 illustrates the potential energy savings of shadow replication over traditional replication for different communication types. As the communication dependencies between processes is reduced the potential energy savings increases. Applications exhibiting full or barrier communication patterns show a savings of 2-11% whereas applications with no communication dependency achieve 23-25%.

The energy savings in a power-constrained environments are greater than those found in non-constrained environments because *shadow replication* enables power constrained environments to have additional main execution nodes, as depicted in Table 7. The reason is that the replica nodes in shadow replication consume less power than those found in traditional replication, freeing up additional power for more nodes.

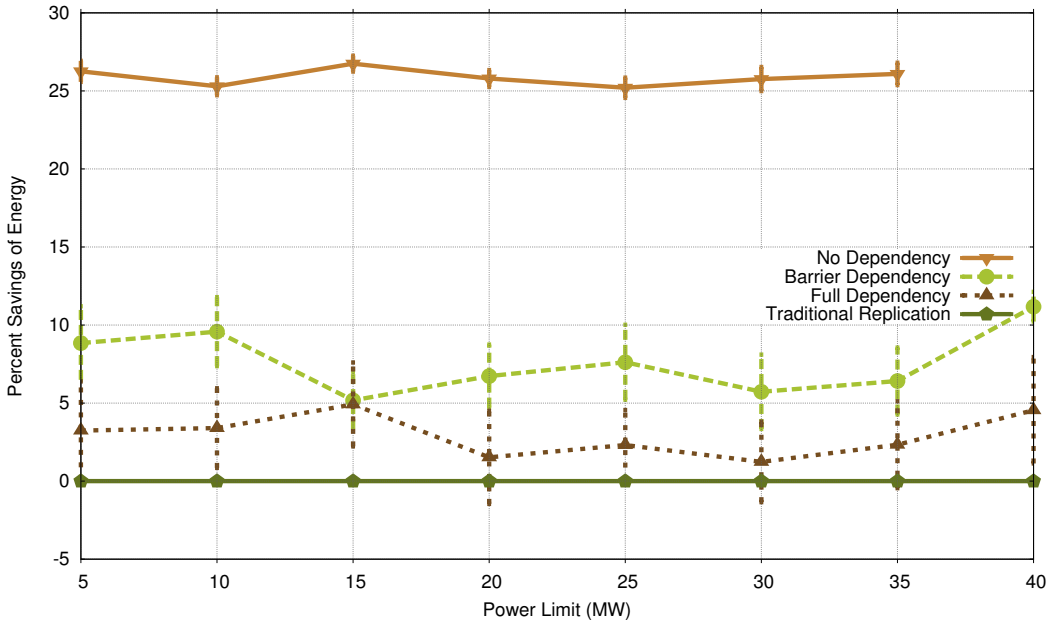


Figure 23: Simulated energy savings of shadow replication over traditional replication, showing different application communication types. Node MTBF=25 Years, Work 3.3 hours per socket for 100,000 sockets, Socket Power Consumption=200 Watts, Static Power 50%, $\alpha = 1.25$

Power Limit	Checkpointing	Traditional Replication	Shadow Replication
5	25,000	12,500	14,611
10	50,000	25,000	29,223
15	75,000	37,500	43,835
20	100,000	50,000	58,447
25	125,000	62,500	73,059
30	150,000	75,000	87,671
35	175,000	87,500	102,283
40	200,000	100,000	116,894

Table 7: Number of sockets available to execute main processes for various fault tolerance methods. MTBF 25 years, Socket Power 200W, Static power 50%.

5.4.3 Sensitivity to Laxity Factor

The simulator has demonstrated that as the laxity factory, α, increases the energy savings acheivable by shadow replication decreases when the application exhibits full or barrier communication dependencies. However, in a power limited environment shadow replicaion does have the ability to save energy however the savings is highly dependent upon the amount of laxity. Figure 24 demonstrates the energy savings sensitity to the laxity factor. From this figure you can see that the maximum energy savings happens when $\alpha = 1.25$ for both full and barrier communication. However, for applications requiring no communication the energy savings continues to increase as the laxity increases. This also demonstrats that as the laxity increases the variablity in energy savings also increase, as indicated by the confidence intervals.

5.4.4 Sensitivity to Socket MTBF

We identified socket MTBF as a major influence of how much energy could be saved using shadow replication. This experiment measures the sensitivity of energy savings over traditional replication as the socket MTBF varies. Figure 25 shows that as the socket MTBF increases the potential energy savings increases for shadow replication, regardless of the applications communication type. However, the socket MTBF has a greater effect on those

applications with communication dependency because as socket MTBF increases the cascading delay effect is reduced.

5.5 CONCLUSION OF SIMULATION ANALYSIS

The simulator was built to further investigate the potential energy savings provided by shadow replication in exascale-class systems. The central conclusion is that shadow replication has the ability to save between 2-25%, dependent upon the application of communication dependency type. As applications have less communication dependency between their processes the potential energy savings of shadow replication increases dramatically - from 2% for application having full communication dependency to 25% for applications with no dependency. Reducing the amount of communication dependency in the application is critical because it allows processes to stop waiting for others to finish. While it might be impossible to reduce this dependency fully any reduction will allow the application to scale better in terms of both fault tolerance and energy consumption.

A secondary conclusion is that shadow replication suffers from cascading delay when communication dependency is introduced. As processes wait on other processes, those waiting processes can also fail, producing a cascading delay that is bounded only by the laxity factor. As the number of sockets increases to the level expected at exascale, cascading delay becomes a significant challenge for shadow replication. To address this concern it is suggested that the laxity factor be below 1.25, which yields the most promising energy savings in our simulations.

Even with cascading delay the simulator demonstrates that in power-limited environments shadow replication can still provide energy savings. This promising result demonstrates that shadow replication still has the potential to save energy and to reduce time to solution in exascale-class systems. The next chapter looks at the viability of implementing shadow replication in actual HPC applications; specifically, application written using the MPI middle-ware layer.

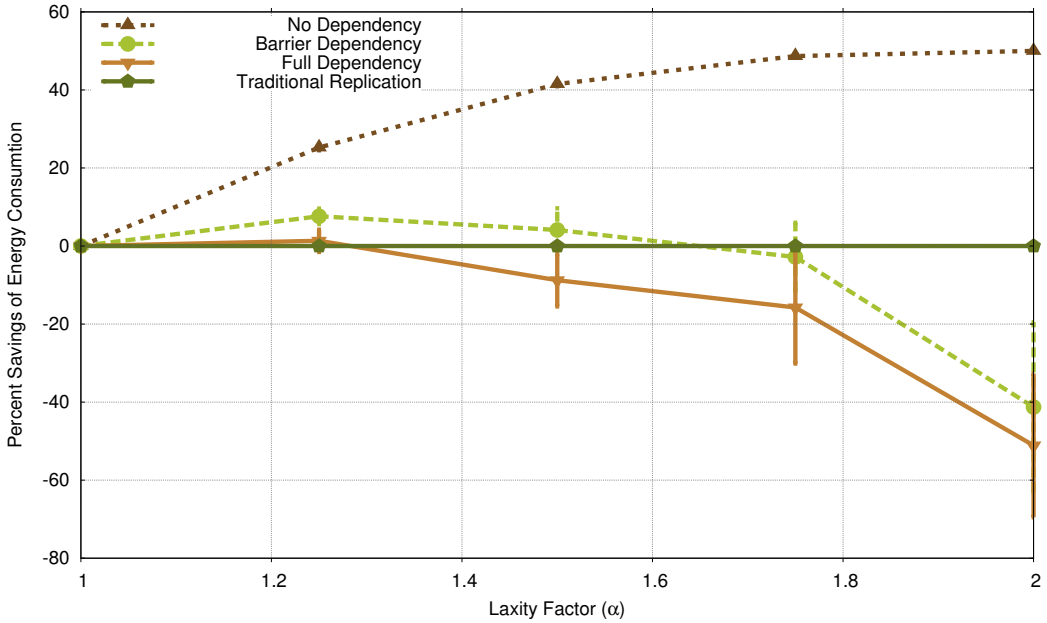


Figure 24: Simulated energy savings of shadow replication over traditional replication, showing different application communication types sensitivity to laxity factor, α. Node MTBF=25 Years, Work 3.3 hours per socket for 100,000 sockets, Socket Power Consumption=200 Watts, Static Power 50%, Power Capacity 20MW

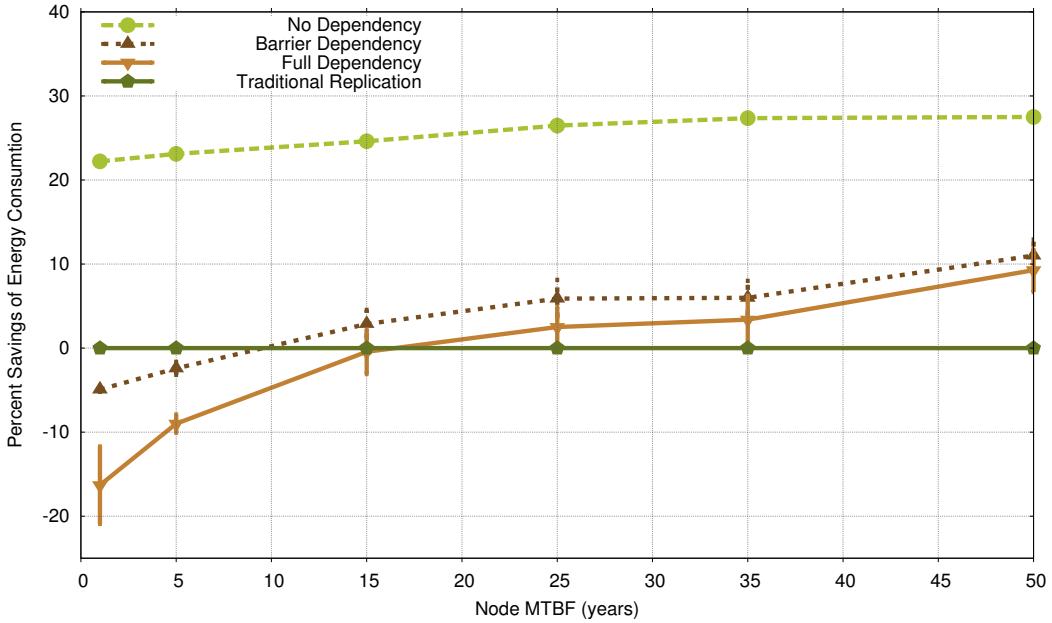


Figure 25: Simulated energy savings of shadow replication over traditional replication, showing different application communication types sensitivity to socket MTBF. Work 3.3 hours per socket for 100,000 sockets, Socket Power Consumption=200 Watts, Static Power 50%, Power Capacity 20MW, $\alpha = 1.25$.

6.0 IMPLEMENTATION

This chapter will provide details of an implementation of shadow replication, called SrMPI, demonstrating that it can execute actual HPC workloads. SrMPI is an implementation of shadow replication for the widely used Message Passing Interface (MPI). We discuss details of the consistency protocols, function modifications, and implementation issues. We then provide an evaluation of the implementation running actual HPC workloads.

6.1 OVERVIEW OF MPI

Message Passing Interface (MPI) is a standardized protocol designed to provide portable message-passing functionality between processes running in a distributed memory system. Although nearly 25 years old, MPI remains the dominant communication mechanism for parallel programming environments and is still found in the largest supercomputing centers. MPI retains its popularity because it enables applications to be written for a wide variety of platforms and systems, resulting in a large library of existing code. As a result of this continuing relevance, we chose to provide an implementation of shadow replication in MPI.

MPI provides both point-to-point communication and collective communication mechanisms, such as broadcast. Point-to-point communication enables two processes, referred to as ranks, to communicate with one another through send and receive functions. There are a series of collective operations that enable group communication between MPI ranks. MPI also provides blocking and non-blocking versions of these communication primitives. Additionally, MPI provides functions for creating process groups which are used during communication. Later in this chapter, details will be presented on how these functions were

modified to support shadow replication.

There are several implementations of the MPI protocol, each optimized for different system environments; however, they consistently adhere to the MPI specifications. This is what has enabled application developers to develop reusable libraries that can be executed in a variety of environments without requiring code modifications. Several of these implementations provide a feature called a profiling library. This feature provides third party libraries the ability to intercept and modify calls to MPI. The profiling library has enabled a variety of packages to be built on top of existing MPI libraries, including replication libraries, checkpointing libraries, and profiling tools.

Moving forward, MPI will undoubtedly be augmented to address scalability concerns of exascale but the core protocol will remain fairly consistent for the foreseeable future. While issues of resilience have long been a topic in the MPI community, efforts to standardize support for resilience have largely failed. Power consumption is becoming a concern and will surely become a topic of interest in future versions of MPI. Another major topic is how to integrate shared-memory parallelism within the context of MPI, which has resulted in pairing MPI with other parallel libraries such as OpenMP and CUDA. The current philosophy is to embrace and extend the protocol to support the growing needs of the community, largely due to the worldwide investment in applications relying upon MPI. Our implementation of shadow replication takes the approach that application code should not have to be modified in order to gain the advantages described in this thesis.

6.2 DESIGN OF SRMPI

The primary purpose of SrMPI is to implement shadow replication within the MPI framework. This involves two main tasks: replicating MPI ranks and adjusting shadow execution speed in response to failures. There are several libraries written to provide replication of MPI ranks; however, none of these libraries provided support for reduced execution speed of the replica ranks. We studied previous implementations of MPI replication [36, 71, 67], and decided it best to implement our own replication library to meet the required consistency

protocols, particularly given that replicas would be running at a slower execution speed.

6.2.1 Consistency Protocols

One of the biggest challenges in developing a replication library is maintaining consistent communication between the main the replica ranks. This is exacerbated by the fact that our replica ranks will be executing at a reduced execution speed. In this section, we will review different consistency protocols and discuss the implementation of those used within SrMPI.

Communication consistency can be maintained either by both the main and replica ranks, or it can be delegated to be done exclusively by the main rank. This distinction is called either active or passive replication. In active replication, the replica ranks actively process every request, whereas in passive replication, the replica follows the main and might ignore or suppress messages while the main node is alive. Active replication requires additional system-wide messages, but it makes error detection much easier. In contrast, passive replication can reduce the amount of communication overhead but will require additional work of the main rank and complicate failure recovery.

SrMPI implements two different active replication designs, called Mirror and Parallel. In mirror mode, the sender rank sends its message to both the main and the replica rank, and conversely a receiver posts a receipt to both the main and the replica sender. This pattern is depicted in Figure 26. To support all possible MPI communications, there is special care needed to handle MPI functions, which will be discussed later in the implementation details. This protocol makes error recovery simple but increases the network bandwidth requirements at each rank.

In parallel mode, the replicas only communicate with the other replica ranks and the main ranks only communicate with other main ranks, depicted in Figure 27. In the event of failure, the non-failed rank begins communicating with the mains and the replicas, effectively switching the failed rank to a "mirrored" mode of operation after failure. A single failure case is depicted in Figure 28. This protocol uses less individual node bandwidth because messages are only sent within their own family of ranks. However, failures must be detected in order to know when to switch into failure recovery mode. This requires additional "short"

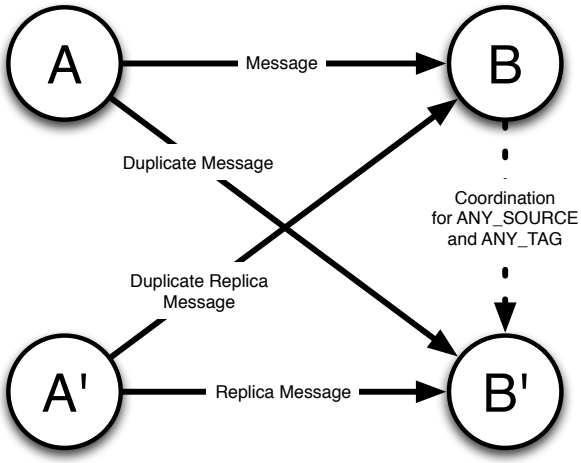


Figure 26: Depiction of communication messages when using mirror consistency protocol.

messages to be sent either between the main and replica ranks or a Reliability, Availability, and Serviceability (RAS) system to detect node failures. Additionally, when a replica rank is promoted due to failure, it must catch up to its faster running main counterparts.

SrMPI does not implement a passive protocol, but there are two different ways that it could be implemented: either the main is responsible for pushing messages to the replica, or the main can wait for receive requests from the replica. This difference is depicted in Figure 6.2.1

6.3 DETAILS OF SRMPI

The SrMPI library is implemented as a profiling library on top of OpenMPI. The first step in the implementation was to modify all collective operations to use point-to-point communication primitives directly. To do this we implemented optimized versions of each collective operation supported in MPI. We did this so that replication could be implemented solely in the point-to-point operations. Next, each point-to-point primitive function was modified to support communication with the replica ranks following the consistency protocols

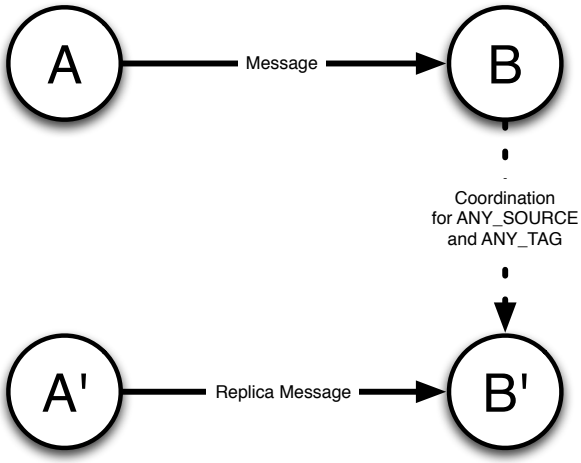


Figure 27: Depiction of communication messages when using parallel consistency protocol.

discussed earlier. Special care is also necessary to support MPI topologies, specifically, Cartesian topology used in MPI applications in the testing testing environment.

When SrMPI is enabled, the `MPI_Init()` function divides the available ranks into two different sets: main and replica ranks. From the applications perspective, `MPI_COMM_WORLD` is cut in half and all associated functions such as `MPI_Comm_size()` and `MPI_Comm_rank()` are modified to return the proper rank and sizes. After the `MPI_Init()` function, the map between main and replica ranks is fixed and available on each rank. By default, the main ranks are the lower half of the rank-space; however, this mapping can be overwritten using a configuration file or environment variables.

Collective operations are written using the point-to-point communication functions; therefore, the implementation only needs to concern itself with the consistency of the point-to-point functions. In mirror mode, each send and receive is posted twice - once to the main and once to the replica. However, we return `MPI_SUCCESS` when either one is successfully returned, both to ease failure recovery and because the replicas will be executing at a reduced speed. In parallel mode, the send and receive only sends their messages to their "rank-family" - either the main set or the replica set.

The implementation assumes application non-determinism can only occur during the

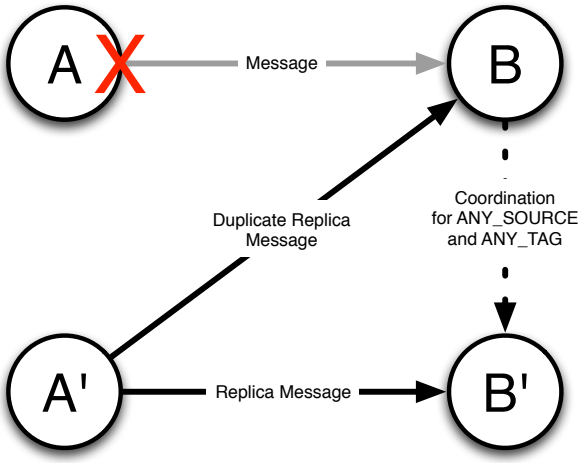
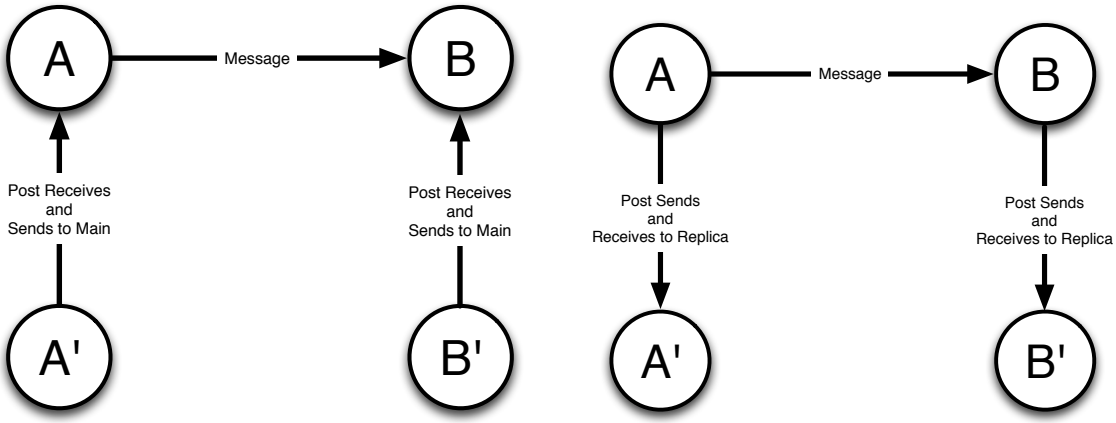


Figure 28: Depiction of communication messages when using parallel consistency protocol and a main failure occurs.

MPI calls and that the application themselves are deterministic. If this was not the case, then application replication would not be possible. The only MPI operations that can result in non-deterministic behavior are non-blocking operations, wildcard receive operations, and `MPI_Wtime()`. To handle these cases, SrMPI assumes the main rank is always the leader rank and determines the outcome of these operations. If the main node fails then the implementation can simply use the result of the replica rank.

For non-blocking receives and sends, which are widely used in MPI applications and collectives, SrMPI must provide mapping between the application request pointers and the multiple requests being maintained by SrMPI. To achieve this behavior `MPI_Wait()` and `MPI_Test()` functions are modified to translate between the actual request pointers and those handled by application. Additionally, in mirror mode the application is allowed to continue once either the main or replica rank has responded. This greatly simplifies fault recovery because the implementation largely stays consistent even in the presence of failures, with wildcard receives as a notable exception.

The other concern is that messages must retain their message order. Because SrMPI posts receives to both the main and the replica ranks a message could be received out of



(a) Replica Pull Model

(b) Main Push Model

Figure 29: Replica Passive Protocols.

order. To ensure this does not happen, SrMPI makes use of an unused tag bit to distinguish between messages received from main and replica ranks. This allows the receive functions to guarantee order even if messages are interleaved between the main and replica ranks.

The MPI protocol allows for wildcard receives, using `MPI_ANY_SOURCE` or `MPI_ANY_TAG`. To ensure consistency, the main and replica ranks must return the same message and SrMPI must then coordinate this between the two ranks. To do so, the main node acts as a leader node and determines which rank or tag will be used, and then informs the replica ranks which message was selected. This requires additional messages to be passed between the main the replica ranks. This poses some challenges when the replica node is running at a reduced speed. In order to overcome this, the messages between the main and replica ranks are performed asynchronously and rely upon the buffering already provided by MPI.

6.3.1 Execution Speed Control

SrMPI uses the findings of the analytical model to determine the execution speeds of the shadow before failure. As discussed in detail in Section 3.3.7, we assume that the main process always executes at the maximum execution speed. The analytical model provided

a closed form solution for the energy optimal execution speed before failure based upon the total amount of work, the node failure rate, and the laxity factor. The MPI layer is unaware of the amount of work necessary, but production scheduling systems, such as SLURM, requires the users to provide an estimated execution time when requesting batch jobs. SrMPI assumes that access to this value is provided and can be used to determine the amount of work necessary. The node failure rate and the laxity factor are then defined as environment variables.

Each architecture will have different available execution speeds and methods for setting and changing those speeds. SrMPI assumes that there is a specified API for both determining available execution speeds and setting those speeds. Most systems have a specified set of execution "gears" which use a combination of both frequency and voltage scaling to achieve an execution speed. However, the analytical model outputs the execution speed in terms of a percentage of the maximum execution speed and is continuous. To translate this into a set of discrete gears, we determine the execution speed which most closely aligns with the available execution gears but is never less than that given by the optimization.

When ranks are initialized, they set their execution speed appropriately, either maximum execution speed for all main processes and the speed before failure for the replica ranks. Once a failure in a main node is detected, the execution speed of the associated replica is increased to the speed of the shadow after failure. In our case this is the maximum execution speed.

In multi-core environments SrMPI must know which core each rank is mapped to on the individual ranks, because DVFS control is typically done on a per core basis. In OpenMPI it is possible to specify the node/core mapping of ranks, and SrMPI relies upon this to know which core is running each rank. In future implementations it would be possible to determine this mapping dynamically, but for this implementation we assume a static mapping is available.

6.3.2 Message Buffers

Within MPI implementations, all communications between ranks happen using a buffer to post both sends and receives. By its very nature, a message is not received until a receive

is explicitly posted to a message buffer. This is ideal for shadow replication because this property allows messages to be posted to message queues and not actually consumed until the process is ready to receive them. Because of this property, a rank executing at a slower speed can receive messages from the faster execution ranks without blocking the faster process. SrMPI exploits these buffers and uses them to buffer messages being queued at the slower execution replica ranks. When the replica rank is ready to process the received message, the rank can simply post a request for that given message.

In mirror mode, the replica process will be able to catch up quickly to the main processes in the event of failure because duplicate messages will be waiting in the message buffer. By contrast, in parallel mode, the replica process will have to wait until all messages are received by the other replica processes.

6.3.3 Function Level Details

In this section we provide details for each function that was modified to accommodate SrMPI.

6.3.3.1 Send In mirror mode, this method sends the message to both the main and the replica task. It then blocks until one of the two messages is successfully received. The other send message remains active but non-blocking until either it is received or `MPI_Finalize()` is called, at which point all pending sends are canceled. If the message is sent from a replica rank, an unused bit in the tag is modified in order to identify the message as coming from a replica. This allows full ordering of the messages on the receive side. In parallel mode, this method sends only one message to either the main or replica rank depending on from what type of node it is being sent.

6.3.3.2 Recv If the source and tag are specified in mirror mode, this method posts two receives: one for the main sender and one for the replica receiver. Once one has been returned, the function returns success to the application. The other receive remains active but non-blocking until it receives a matching message or `MPI_Finalize()` is called, at which point all pending receives are canceled. In parallel mode, this method only posts a receive

request for the main or replica rank depending on the node type from which it was called.

If the source or tag is not specified and the main and the replica rank are still active, the wildcard coordination protocol is invoked. In this protocol, the main is selected as the leader and will first determine which source and/or tag is selected, after which time the main will inform the replica of the selected source and/or tag. Subsequently the replica will post a receive for the selected source and/or tag that the main has already received. Additional details were presented in the implementation details.

6.3.3.3 iSend This method implements the same semantics of blocking send function except instead of blocking, it returns a request pointer to the application. This request pointer will then be used by the application in either the `MPI_Wait()`, `MPI_Wait()` or `MPI_Test()` functions to determine when one of the sends completes. Just as in the blocking version, this version will return success when one of the sends completes, and the other send will stay active until it succeeds or `MPI_Finalize()` is called. Again, for parallel mode only one send is posted.

6.3.3.4 iRecv This method implements the same semantics of the blocking receive function, except instead of blocking this method returns a request pointer. This request pointer will then be used by the application in either the `MPI_Wait()`, `MPI_Wait()` or `MPI_Test()` functions to determine when one of the sends completes. Just as in the blocking version, this version will return success when one of the receives completes, and the other send will stay active until it succeeds or `MPI_Finalize()` is called. Again, in parallel mode only one receive is posted.

This function is further complicated when the source or tag is unspecified, although the same semantics are implemented in the non-blocking version as were implemented in the blocking version.

6.3.3.5 Test, Wait These functions were modified to use a modified request pointer that the application is given by the non-blocking send and recv methods. This method then looks up the actual request pointers which were issued by SrMPI and uses them to determine how

to respond these functions while maintaining the MPI semantics.

6.3.3.6 Topology Functions These functions had to be modified because the number of ranks and rank ordering is changed and managed by SrMPI. Specifically when creating a Cartesian topology the ranks must be ordered consistently to ensure that any application using these methods received consistent results.

6.4 EVALUATION OF IMPLEMENTATION

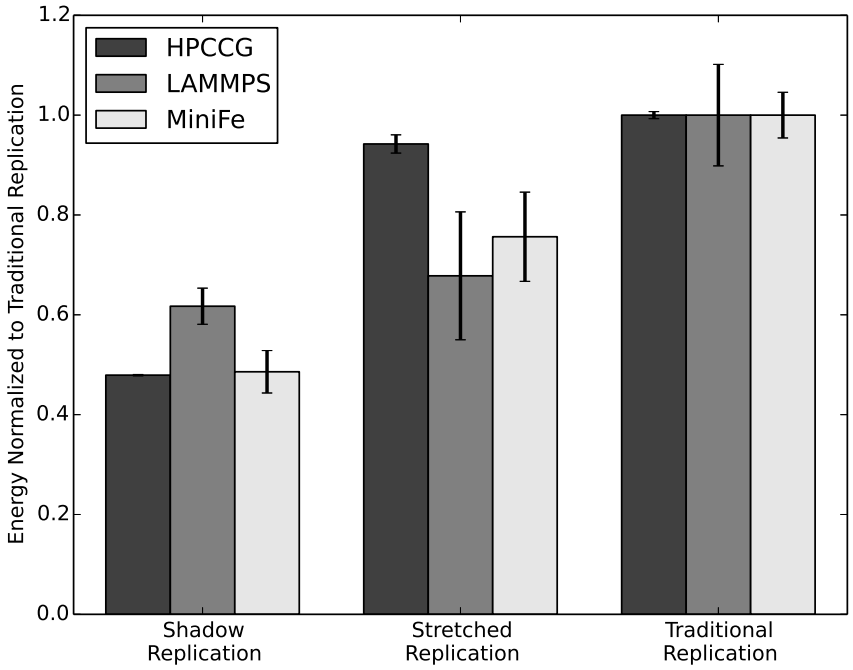


Figure 30: Experimental results of the energy savings achieved by different replication schemas, executing on 16 cores.

The purpose of these small-scale experiments is to demonstrate that the power-aware replication techniques can provide measurable energy savings for actual HPC applications.

Application	Replication Method	Energy	Time to Solution
LAMMPS	Traditional	1.000	14.0 min 0.0 sec
LAMMPS	Stretched $\alpha = 1.5$	0.678	14.0 min 37.0 sec
LAMMPS	Shadow $\alpha = 1.5$	0.617	18.0 min 56.0 sec
HPCCG	Traditional	1.000	2.0 min 5.0 sec
HPCCG	Stretched $\alpha = 1.5$	0.942	2.0 min 11.0 sec
HPCCG	Shadow $\alpha = 1.5$	0.479	2.0 min 14.0
miniFe	Traditional	1.000	1.0 min 42.0 sec
miniFe	Stretched $\alpha = 1.5$	0.756	2.0 min 10.0 sec
miniFe	Shadow $\alpha = 1.5$	0.485	2.0 min 9.0 sec

Table 8: Experimental data.

In Figure 30, the average total energy consumption of multiple application runs are shown for each replication technique, normalized to the energy consumed by traditional replication. This shows that power-aware replication techniques reduce overall energy but also demonstrates that the amount of savings is application dependent, as previous studies have found [64]. HPCCG and miniFe show the maximum energy savings. This is because they are simple applications that are processor bound. Looking at LAMMPS, which is a production application, one can see that the energy savings follows the same trend but also that the amount of energy saved is less than for the mini-applications. While it is hard to predict exactly what the energy savings will be, it is clear that our proposed techniques have the potential to save energy.

In Table 8 we present the energy consumption normalized to the pure replication energy and the corresponding time to solution of each of our configurations. Determining the execution speeds based upon α is not that accurate. As previous studies [64] have shown, applications are affected differently by CPU scaling. In LAMMPS, reducing the processor speed by a third had very little effect on the time to solution for the application. By contrast, shadow replication causes LAMMPS to slow down significantly. We believe this is because LAMMPS makes use of ANY_SOURCE receives, which causes a blocking dependency between the main and shadow processes, slowing down the overall execution.

To confirm our assumptions about overhead power, we looked at the component level energy usage over runs of real applications. In Figure 31 we show the percentage of energy

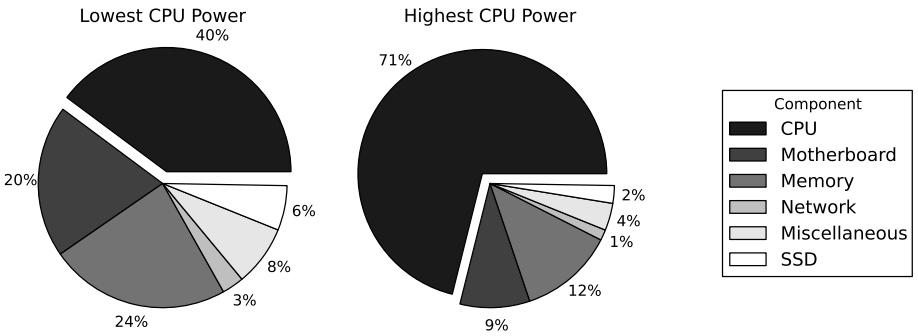


Figure 31: Component level energy usage for LAMMPS

consumption by component for multiple runs of the LAMMPS application. The first chart is the energy consumed when running LAMMPS at the lowest possible execution speed. In this case, the CPU consumes 40% of the overall energy. The second chart shows the energy consumption when running at full power and the CPU consuming 71% of the overall energy. From this, the estimated amount of overhead power is 67%. We observed a similar pattern for other applications, concluding that overhead power in our system is 60-67%.

6.5 SUMMARY OF MPI IMPLEMENTATION

By implementing shadow replication within MPI, this chapter has confirmed its feasiblity to provide fault tolerance in high performance computing. Additionally, this chapter discusses some challenges faced within the MPI middle-ware and how they were overcome in this implementation. To demonstrate the prototype implementation this chapter presents the energy savings achieved using SrMPI using an actual HPC workload.

7.0 CONCLUSION AND FUTURE WORK

The main goal of this thesis is to explore the possibility of building a fault tolerant model for future exascale-class systems that is more energy efficient than existing fault tolerance models. To achieve this goal, *shadow replication* a power-aware fault tolerant computational model for failure-prone, power-constrained exascale HPC environments is developed. Using analytical models and simulation studies, the ability of shadow replication to conserve energy in power-constrained environments, while meeting the time to solution requirements, has been demonstrated. Furthermore, the practicality of shadow replication is demonstrated thorough the implementation of this fault tolerant computation model in the Message Passing Interface (MPI), a widely used middle-ware for process communication and synchronization in HPC environments. The insights gained in developing shadow replication paved the way to an easily implementable, power-ware technique that when incorporated into coordinated checkpointing fault-tolerant models, leads to energy savings in today's systems. The experimental results show that the levels of energy savings achieved by these methods could be significant. This final chapter will summarize the contributions of this dissertation and discuss possible future work.

7.1 CONTRIBUTIONS

The research work presented in this thesis, backed by analytical and experimental results, proves the feasibility of the a power-aware tolerant model, shadow replication, that can achieve substantial levels of energy savings, while providing the same level of performance and resilience, in terms of adhering to time to solution requirements and power constraints.

The implementation of shadow replication in MPI further demonstrates how this model can be incorporated in the most commonly used fault tolerant middle-ware in HPC environments. The experimental studies carried out in this thesis demonstrate significant levels of achievable energy savings in power-constrained HPC environments. The major contributions of this work are discussed below:

Computational Model for *Shadow Replication*. In order to investigate the behaviors of shadow replication, a computational model was defined and the possible techniques to minimize energy consumption were explored using this model. The resulting model defines a duplex system, where two processes execute the same computation, with the main/primary process executing at a higher execution speed than the replica/shadow process. Upon failure of the main process, the shadow process increases its execution speed in order to achieve the targeted response time of the task. The potential power savings achieved by the model, are a direct result of executing the shadow process at a slower speed; consequently, shadow replication consumes less power than traditional replication. The hypothesis is that this computational model would be more energy efficient than replication and traditional rollback-recovery methods, especially in future exascale environments where system failure rates are expected to be high.

Analytical Energy Models for Fault Tolerant Mechanisms. In order to evaluate the energy consumption of shadow replication, we developed an analytical model for both replication and coordinated checkpointing. An analytical framework is developed, assuming an exponential failure model, to compute the expected energy consumption of fault tolerance models.The replication model is flexible enough to describe a variety of replication techniques, including shadow replication, stretched replication, and traditional replication. Using Daly's model [23] for estimating time to solution for coordinated checkpointing, we define an expected energy model.

Analytical and Simulation Analysis of Energy Consumption. The original analytical model was focused on deriving the speeds of execution of the main and its associated shadow, both before and after failure. The objective is to gain understanding of the basic behavior of the model and determine the challenges and research directions that must be addressed for a practically feasible, easily implementable model in exascale systems. As

such, the original model does not fully capture the communication intricacies of exascale applications. Unfortunately, augmenting the original model to account for the complexity of communication patterns and requirements in HPC environments makes the model mathematically intractable. To accurately capture the different types of communication patterns in HPC environments, a simulator was built to emulate three types of communication dependency behaviors, namely no communication, barrier dependency and full dependency. Using the simulator, a study was carried out to measure the levels of energy savings achievable by shadow replication. The analysis shows that, depending upon the application and system characteristics, *shadow replication* has the ability to save 2-47% of the energy consumed by applications in power limited environments.

MPI Implementation of Shadow Replication. In order to prove that shadow replication is feasible in high performance environments, we then presented a prototype implementation in the message passing interface (MPI), which we called SrMPI. The thesis presents details of the implementation and discusses the challenges that needed to be overcome during implementation. Selected small-scale experimental results were presented using SrMPI, demonstrating its potential to save energy in the non-failure case.

In the Apendix is the related contribution of exploring the power analysis of checkpointing techniques. In this section, we present component-level power data for coordinated checkpointing and suggestions for how to save energy during the writing and restoring of checkpoints. This analysis shows that during IO bound operations there is the potential to reduce the energy consumption by reducing the frequency and/or voltage of the CPU using dynamic voltage/frequency scaling (DVFS). This analysis also discusses the interplay between the processor and the interconnect framework providing the stable storage being used by coordinated checkpointing.

7.2 FUTURE WORK

Expanded Computational Models. This thesis has provided justification for the fact that power-aware fault tolerance methods can not only save energy but might also provide

faster time to solutions. However, while *shadow replication* shows promising energy savings in exascale-class systems, the optimality of such a model for different failure models and communication patterns is yet to be proven. The model presented in this thesis showed that cascading delay can result from successive process failures, leading to adverse performance implications. Additional research is required to develop models to overcome cascading delays and continue to provide energy savings.

Jumping Shadows. Due to the reduced speed of the replicas, one concern is the growth of the message queues between the replica and the main process. The rate of growth depends upon the application message rate, as shown in Section 4.1.4. One possible solution to limit the message log growth, is to synchronize the shadow processes such that message buffers are flushed before they reach their capacity. This would require either monitoring of message buffers or the development of a predetermined schedule which would drive the main process to synchronize with its associated shadows processes, and allowing shadows to "jump" to the current main execution point. By copying the state of the main process to all the shadow processes, buffer overflow can be prevents, while maintaining the same level of fault tolerance. This is similar to independent checkpoints, which has been proposed in uncoordinated checkpointing, and could potentially making "jumping shadows" a general solution to the message log growth problem. Another approach would draw upon existing methods used to address the message log problem in uncoordinated checkpointing, such as send-determinism [45].

Shadow Replication in Cloud Computing. One of the hallmarks of the cloud computing environment is its support of decoupled applications, which allows massive parallelism. This is ideal for tasks requiring timely, but not necessarily precise, responses such as indexing content and providing shopping suggestions. The inherent laxity in precision allows massively parallel applications to continue to respond in the face of failures and other system events. As these solutions are applied to applications traditionally dominated by centralized database servers, such as medical analysis and banking, higher accuracy and precision may be needed, while still achieving fast responses. This will result in tightly coupled real-time applications being executed in the inherently unreliable cloud environments, which is similar to those found in HPC environments. Applying *shadow replication* to cloud environments

has the promise of being able to demonstrate the same energy savings we have seen in this thesis. Our group has already began to explore *shadow replication* in the cloud; however, additional work is still necessary to conclusively demonstrate its merit [21].

Interplay between Fault Tolerance and Power. This thesis presents additional evidence that addressing fault tolerance and power management in tandem results in more efficient designs both in future systems and production systems executing code today. Through the literature review we have found that very little is known about the interplay between resilience, faults, power, and energy, especially in large-scale distributed systems. While this is a broad area of research, it is believed that through understanding the intricacies of these relationships new models for resilience and power management will become apparent.

QoS-based Resilience. In this thesis we proposed using available slack in time to dynamically trade off the hardware and power resources necessary to provide fault tolerance. In a broader sense, target time to solution is simply a way of defining a required quality of service (QoS) for the application. Using this reasoning it would be possible to use other metrics of application QoS to provide the necessary "slack" to achieve fault tolerance. HPC Applications often have the ability to adjust the fidelity of their results, such as the granularity of a simulation or the precision of convergence. If the fault tolerance system had the ability to adjust the applications fidelity at runtime, it could harness that leverage to create "slack" while meeting the applications defined QoS in faulty environments.

To accomplish, this we are proposing a new computational model, called *shadow computing*, which provides goal-based adaptive execution to meet the requirements of complex applications executing within complex systems. Adaptive execution is the ability of the system to dynamically harness all available resources to achieve the highest level of QoS for a given application. Additionally, the application will have the ability to adjust the quality of its results at runtime and the system could do so to be able to provide fault tolerance while meeting the applications stated QoS. Similar to *shadow replication* fault tolerance would be provided by using *shadow* processes but instead of executing at reduced execution speed, they would be executing at a reduced quality metric, such as fidelity. This technique has some precedence in networking but to our knowledge has not be explored in the context of distributed systems.

Mobile Distributed Computing Environment. As processing power continues to increase on mobile devices, there will be an opportunity to perform computations in a cluster of mobile nodes. This ushers in an extreme version of moving the computation to the data, in which the nodes collecting the data would collaborate with one another, in physical proximity, to perform complex computations that any single node would be unable to complete by itself. Such an environment would be plagued with reliability concerns and will require new programming models and systems software to provide overall *system resilience* to deliver the required QoS to supported applications. We believe that *shadow computing* has potential to provide fault tolerance in such environments.

APPENDIX

POWER MEASUREMENT OF CHECKPOINTING

In this thesis we have demonstrated that there are power-aware alternatives to checkpoint/restart that should be considered for exascale-class machines. However, fault tolerance in today's large scale production systems continues to rely, almost exclusively, on coordinated checkpoint/restart. This appendix will look closely at the energy and power consumption of checkpoint/restart and investigate methods for reducing overall energy consumption without changing the behavior of checkpoint/restart.

As previously discussed, during normal operation, checkpoint/restart (or *rollback recovery*) protocols [31], periodically record the state of all application processes to stable storage (the checkpoint stage). When a process fails, a new incarnation of the failed process is *recovered* from the most recent checkpoint (the restart phase). This limits the amount of lost work to only that since the last checkpoint (the rework stage).

The prevalence of checkpoint/restart is due to a number of factors: historically failures have been relatively rare events; applications are generally self-synchronizing; and application state can be saved and restored much more quickly than a given system's mean time to interrupt (MTTI). All of these factors have kept the overheads of traditional checkpoint/restart on current systems limited to a modest portion (currently perhaps 10-25%) of an applications total time to solution. For future exascale-scale systems, a number of these assumptions may change such that the overheads of traditional checkpoint/restart could become prohibitively expensive [82, 35, 101].

Unreliable systems are nothing new. For example, ASCI White originally had a Mean

Time Between Failure (MTBF) of only 5 hours [102]. This was subsequently improved to approximently 50 hours. It is expected, however, that exascale-class systems will experience failure rates significantly higher than those observed in the highest failure prone HPC systems to date. Worse than that, the methods to alleviate the impact of failures on system performance, not only add overhead, but also increase power consumption. Figure 32 illustrates the problem: The time to checkpoint and restart increases exponentially with system size. These are high-power operations and, therefore, energy consumption will also increase at a much higher rate than the increase in number of sockets would suggest.

A number of recent studies show that general power consumption can be reduced during writing of a checkpoint to stable storage [96, 74]. The CPU is the largest consumer of power on an HPC node, but its power consumption can be controlled using Dynamic Voltage and Frequency Scaling (DVFS). The prior work suggests that during the I/O intensive checkpoint and restart operations, throttling the CPU can save power without impacting checkpoint performance.

In this work, we build on previously published framework, examining component-level power consumption to measure the energy cost of checkpoint operations to local SSD's and remote storage [65]. For remote operations, we compare the power consumption of IP and RDMA, both over an InfiniBand network. With these baselines in place, we then use DVFS to throttle CPU speed during checkpoint writes to measure the energy savings and performance impact. We find that overall energy savings of 10% are possible with actual HPC workloads. Furthermore, the choice of network protocol and local versus remote storage have important energy consumption impacts that needs to be considered when designing fault tolerance protocols that make use of hierarchical storage.

A.1 EXPERIMENTAL RESULTS

Before we present the results of this experimental study, we explain our experimental methodology and the framework used to carry out the study. We then measure the effect of lowering the frequency and voltage of the CPU during a checkpoint write operation to local and re-

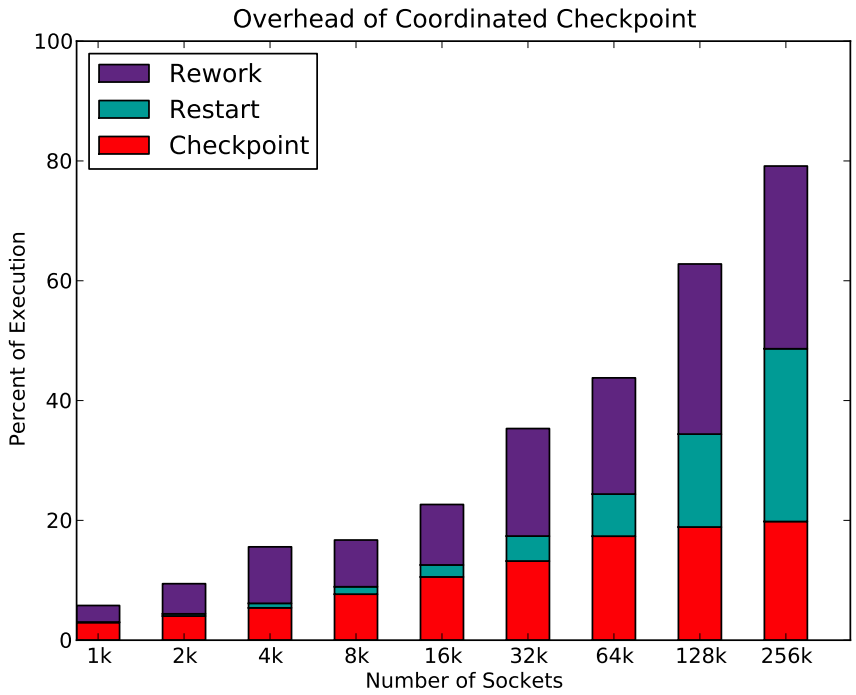


Figure 32: Percent of wall-clock time spent in each coordinated checkpointing component using a validated simulator [93]. Checkpoint commit time is 15 minutes, a value shown seen on many extreme-scale systems [12, 39]; and a node MTBF of 15 years [101], using the optimal checkpoint interval from Daly [22].

P-state	Volt	Frequency
P0	1.363	3800
P1	1.288	3400
P2	1.200	2900
P3	1.075	2400
P4	0.963	1900
P5	0.925	1400

Table 9: Software visible power states for x86 cores on AMD K10-5800K.

mote storage. For remote I/O over InfiniBand, we switch between the IP and the Remote Direct Memory Access (RDMA) protocol and evaluate each. We then repeat these three experiments for two applications and analyze the results obtained in each case.

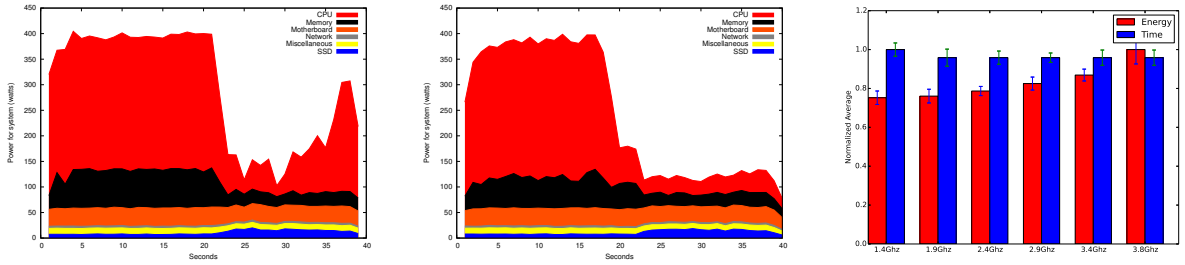
A.1.1 Methodology

We used DVFS to vary the CPU frequency and voltage during a checkpoint write to determine the potential energy savings available. Using DVFS, several different discrete "gears" are available for CPU frequency; the specific "gears" that are available on the testbed hardware and explored are listed in Table 9. We gathered component-level power consumption data from several nodes; the data gathering was accomplished while writing checkpoints to local and remote storage. For remote access, we used two NFS solutions: one using the kernel network stack and the other using the IB RDMA interface.

A.1.2 Experimental Framework

We measured power consumption on a cluster with 104 nodes, each equipped with an AMD Llano Fusion APU, which is a 4-core AMD K10 x86 paired with a 400-core Radeon HD 6550D. In the experiments, we only use the x86 cores, ignoring the available GPU. The CPU frequency and voltage are modified using the powernow_k8 kernel module. There are six available gears ranging from a frequency of 3.8 GHz down to 1.4 GHz.

Component level power measurements of the CPU, memory, on-node SSD device, motherboard and the Qlogic QDR InfiniBand HCA were performed using a custom designed



(a) 3.8 GHz (b) 1.4 GHz (c) Time vs Energy

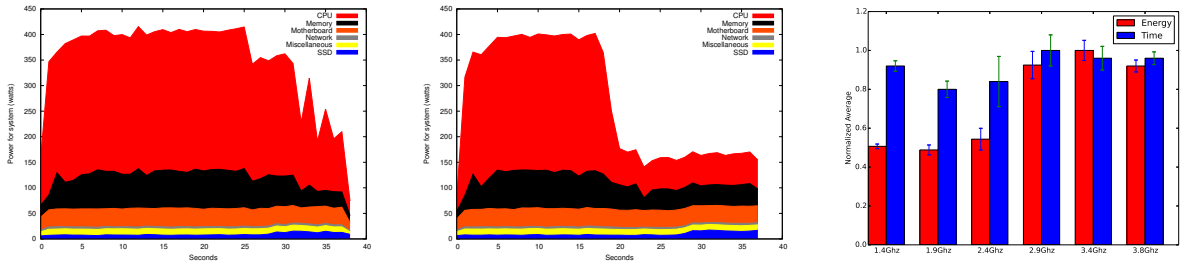
Figure 33: *Local SSD*. Component level power profile during a coordinated checkpoint of HPCCG in a 4-node cluster using 16 processes, each process checkpoint was approximately 1.5GB. The Time versus Energy plot shows the energy and time to complete the checkpoint operation over 10 separate runs. Error bars represent standard deviation.

power measurement system. More detail about the system is available in [65].

We used LAMMPS [87], a molecular dynamics code, and HPCCG, a conjugate gradient solver from Sandia's mantevo suite [97] as the MPI applications for the experiments. These applications together expose a range of computational behaviors, ranging from mixed IO and computing as exhibited by LAMMPS and compute intensive behavior typical of HPCCG. LAMMPS is a production level code that is frequently run at very large scales on U. S. Department of Energy leadership class systems, while HPCCG is a mini-app that is representative of a real, finite element code. Both used Open MPI and the built-in BLCR [47] support for checkpointing.

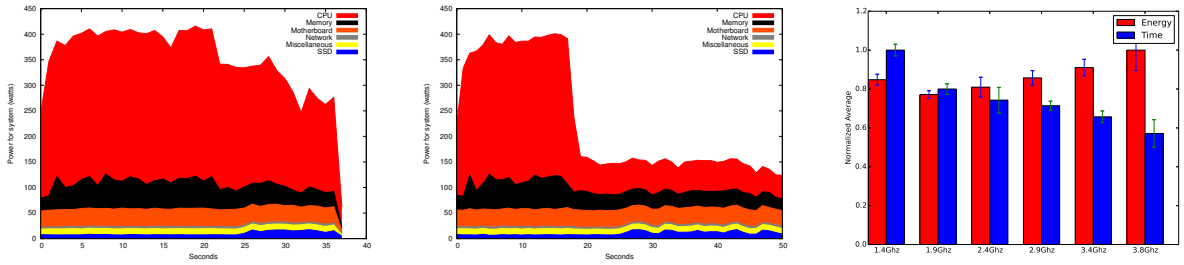
A.1.3 Local Checkpoint Power Profile

Checkpointing is an I/O intensive operation and previous work has indicated that CPU utilization is low during a local checkpoint [96]. This section explores what energy savings may be possible when checkpointing to a local SSD. Figure 33(a) shows the component level power profile of 4 nodes running at full processor speed performing a local coordinated checkpoint. As the processes pause their execution, a drop in the amount of power consumed by the CPU



(a) 3.8 GHz (b) 1.4 GHz (c) Time vs Energy

Figure 34: *Remote SSD using IP over InfiniBand.* Component level power profile during a coordinated checkpoint of HPCCG in a 4-node cluster using 16 processes, each process checkpoint was approximately 1.5GB. The Time versus Energy plot shows the energy and time to complete the checkpoint operation over 10 separate runs. Error bars represent standard deviation.



(a) 3.8 GHz (b) 1.4 GHz (c) Time vs Energy

Figure 35: *Remote SSD using RDMA over InfiniBand.* Component level power profile during a coordinated checkpoint of HPCCG in a 4-node cluster using 16 processes, each process checkpoint was approximately 1.5GB. The Time versus Energy plot shows the energy and time to complete the checkpoint operation over 10 separate runs, error bars represent standard deviation.

occurs, even without modifying the operating voltage or frequency of the processor. This initial result underscores the opportunity to save energy by reducing the clock frequency and voltage of the processor.

Observe in Figure 33(a) that even though CPU power consumption drops during a checkpoint, the CPU is still the dominant consumer of power. All other components consume less than half the power consumed by the CPU. This makes CPU power an excellent candidate for energy savings during checkpoints. SSD power consumption does increase during the checkpoint, but it is not as promising a candidate for energy savings as the CPU as the percentage of energy consumed by the SSD is far less than that of the CPU.

Figure 33(b) shows the result of checkpointing with the processor frequency set to 1.4 GHz, the lowest possible gear. Reducing the CPU frequency (and voltage) reduces total energy consumption and smooths out power consumption during the checkpoint time. During the I/O operation, there are times that the CPU is blocked in a busy loop waiting for the I/O to complete. This polling for I/O completion can lead to small power spikes. By reducing the CPU frequency, these spikes occur less frequently because the CPU is less likely to block on I/O.

Reducing the CPU frequency during checkpoint causes the operation to take slightly longer. In Figure 33(c), we compare the consumed energy during the checkpoint time and the total execution time for the operation. We show the results for all 6 available voltage and frequency gears in our environment. To compare both time and energy in the same figure, we normalized the values to the highest measured. This figure confirms that during extended local I/O operations, reducing the CPU power has little effect on the time to completion, but can save up to 25% of the total energy.

These results are very encouraging, but for a checkpoint to be usable it must be stored on a device that is failure independent of the node performing the computation. In practice this means that writing a checkpoint also includes a network operation, to copy or stream the checkpoint data to a network storage device. The next section will explore the power profile of writing a checkpoint to a network device.

A.1.4 Power Consumption by Network Type

Checkpoints are clearly I/O bound operations, and checkpointing to a remote location will therefore be network intensive. CPU involvement in the network operations is highly dependent upon both the software and hardware being utilized. If network operations require significant CPU resources, reducing processor frequency can have a significant effect on the time necessary to write a checkpoint. This will impact the potential energy savings for checkpointing over a network. In order to study the effect of distributing checkpoints across a network we chose to store checkpoints in neighboring compute nodes. Because this is a coordinated checkpoint there should be no interference with the MPI application being checkpointed. We tested two different network configurations using NFS: IP over InfiniBand and RDMA over InfiniBand.

Both of the network configurations tested use InfiniBand network hardware. InfiniBand networks can be implemented as an offloaded or onloaded, or partially onloaded and offloaded solution. The results presented in this thesis measure energy consumption of a system with a partially onloaded InfiniBand Qlogic host channel adapter (HCA). With a fully offloaded InfiniBand HCA, such as those from Mellanox, the CPU utilization during the checkpoint could reasonably be expected to be lower, and therefore greater savings may be possible.

A.1.4.1 IP over InfiniBand IP over InfiniBand (IPoIB) is a protocol that allows encapsulating IP packets for their transmission over InfiniBand network hardware [20]. This requires mapping IP addresses to InfiniBand subnets that support IPoIB. The underlying network driver/hardware is an InfiniBand HCA, which transmits the encapsulated packets inside of native InfiniBand messages. IPoIB utilizes portions of the kernel IP networking stack, and associated upper layer transports (e.g. TCP/UDP). Therefore, the performance benefits of OS bypass is not available to IPoIB applications and CPU load is increased over native IB. IPoIB therefore provides the convenience of a socket interface to an application but with the drawback of a performance penalty.

A.1.4.2 RDMA over InfiniBand Remote Direct Memory Access (RDMA) is a key feature of InfiniBand networks. It allows for a source node to transmit data directly into a target node's memory. There are two methods for performing RDMA. The send/recv method uses target side "recv" queue (RQ) entries that are matched to incoming messages. These RQ entries indicate where a given message should be placed in memory, which can be an application's buffer, avoiding any intermediary copies that would otherwise be performed in a typical kernel network transport communication.

The other method of performing RDMA is the Write/Read approach, which has the source node include all of the information on where the data is to be placed in the target node's memory. This requires that the source node has knowledge of the target node's memory, including what areas are designated for that source node's messages. This is typically accomplished through an exchange of data prior to RDMA communication, or through buffer advertisement while communication is ongoing.

RDMA can provide very low latency networking, and small message RDMA operations can have sub-microsecond latencies, while large messages can have very high throughput.

A.1.4.3 Remote Checkpoint Power Profile Figure 34 shows the power profile of writing a checkpoint over a network using IP over InfiniBand. When executing the checkpoint at full speed there is significantly more CPU activity than that observed for local SSD checkpoints (Figure 33). This increased CPU utilization is due to the network stack processing required by our onloaded InfiniBand hardware. Reducing the CPU speed reduces the energy consumption significantly while even potentially increasing checkpoint performance. This result is encouraging and shows that reducing CPU power can result in energy savings with little additional overhead, and in the case where resource contention was causing slowdown, actually increase checkpoint efficiency.

It would be reasonable to expect that a local SSD checkpoint would be faster than a network operation, however we consistently saw full speed IP over InfiniBand and RDMA outperform local SSD writes, albeit only by a small amount. The reason for this unexpected result is the buffering behavior of NFS. Due to it's write buffers, NFS reports to the client that the write operation is complete as soon as the entire message has been buffered at the server.

The actual write to disk then finishes, allowing the client to proceed with computation. This results in the network copy appearing to be slightly faster than the local SSD write, as local SSD write-caching is not available in the Linux kernel we used for testing. As can be observed in 34(c), reducing the CPU power during checkpoint operation can save 50% of the consumed energy. Further research is necessary to determine the impact that the NFS buffered writes might have upon energy and time to solution.

In contrast, Figure 35(a) shows a near constant use of the CPU during the RDMA network transfer. Although, in principle, RDMA is OS bypass and can be offloaded, our interface cards make heavy use of the CPU during RDMA transmission. With an unthrottled CPU, the transmission is slightly faster than IPoIB. Reducing the CPU speed we observe that RDMA takes significantly longer than it did at 3.8 GHz. Figure 35(c) shows that while we can save 15% of the energy, this causes the checkpoint time to nearly double.

A.1.5 Checkpoint Energy over Application Execution

In the previous sections we examined the power profile of a single checkpoint event during the execution of HPCCG. In this section, we look at the power profile over three checkpoints taken during a run of LAMMPS using the Lennard-Jones workload.

Figure 37(a) shows the power profile of LAMMPS when the three checkpoints go to the local SSD. During the local checkpoints, the CPU power consumption is considerably reduced. However, when writing remote checkpoints, shown in Figure 38(a) and 39(a), the CPU power is much higher during these times. With reduced CPU frequency shown in Figures 37(b), 38(b) and 39(b), we see a drop in power consumption and an increase in execution time, particularly for the off-node operations.

By focusing on the checkpoint event itself in previous experiments, we were able to draw some conclusions regarding the use of DVFS during those events. However, when evaluating the energy and time to solution we found that the variance of the application runtime was too high to draw any conclusions. This was unexpected as LAMMPS typically has very little variance in execution time. Further investigation determined that the variance was introduced by a helper thread for Open MPI's BLCR support. Figure 36 shows the effect of

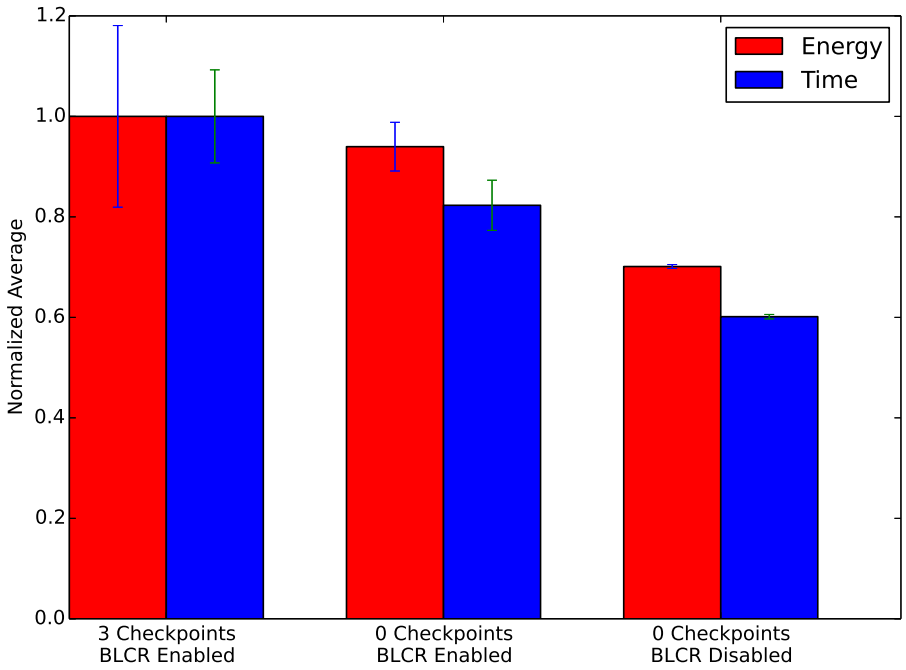


Figure 36: Time to solution and energy consumed for LAMMPS using different configurations. These experiments are ran using Open MPI 1.3.4.

BLCR support in Open MPI on LAMMPS runtimes, even when no checkpoints are taken.

The amount of variance is further magnified when checkpoints are actually written. This is because the time to coordinate the checkpoint is dependent upon how far into the application execution it is requested. The variance introduced by Open MPI BLCR support moves this request from run to run and obscures conclusions. Therefore, we do not show energy graphs for the overall application run.

We can, however, conclude from the power profiles that the behavior during the checkpoint event is the same even when executing multiple checkpoints over the execution of the application. Because these profiles show a similar behavior to that found in the HPCCG experiments, energy savings are expected for the overall application execution. Future work must address the variance introduced by the checkpointing support in Open MPI to be able to confirm this conclusion.

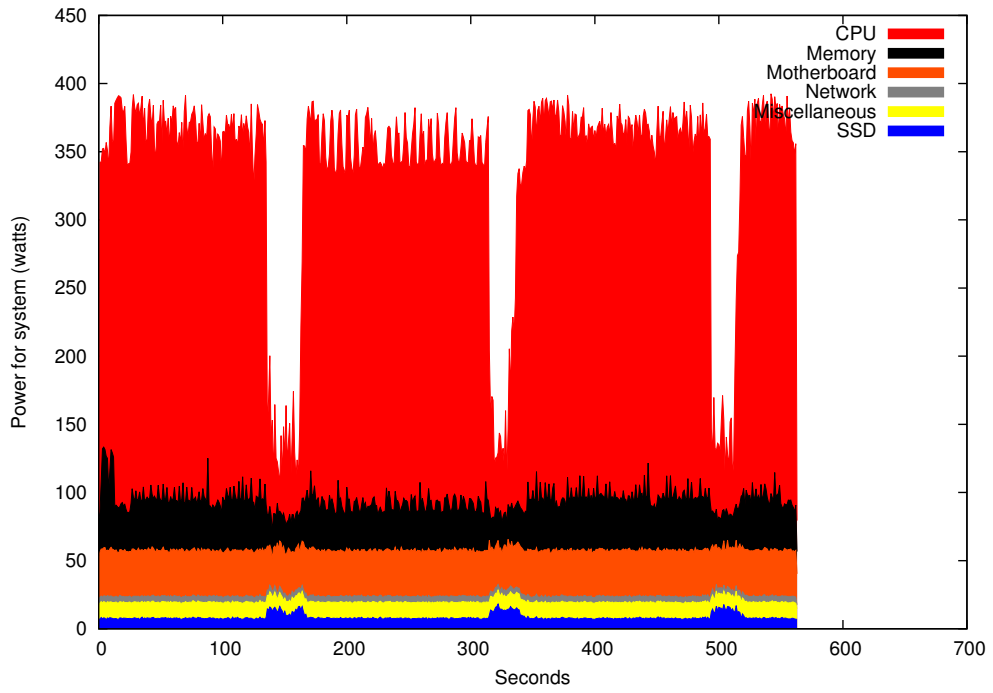
A.2 CONCLUSTIONS OF POWER MEASUREMENT OF CHECKPOINTING

This chapter demonstrates there is potential for realizing energy reduction during checkpointing events using DVFS – all while having little to no impact on checkpointing performance. More specifically, we show a maximum 50% energy savings by throttling CPU power consumption during I/O intensive checkpoint operations. Given that these checkpoint operations can consume 20% of an applications total runtime (see Figure 32), this leads to a possible 10% total application energy savings from DVFS with checkpointing. We also show that this potential is highly dependent upon the network characteristics. For the Qlogic InfiniBand cards in our test cluster, the opportunity to save energy is small compared to the benefits seen committing checkpoints to local storage due to network onload versus offload issues. The onloaded interface used showed IPoIB, while slower than RDMA, has greater potential to save energy during large I/O operations. The result of our experimental study, the first of its kind to explore DVFS techniques using IP over InfiniBand and RDMA during checkpoint operations, is in conflict with general conclusions reported in existing lit-

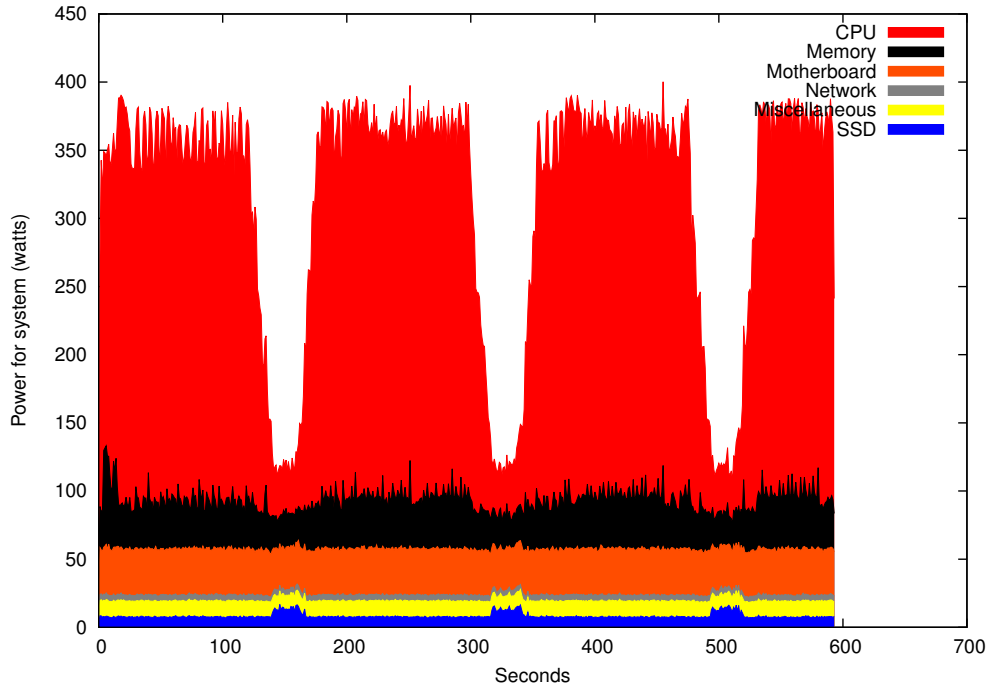
erature [**?**]. In these studies the focus has either been on experimental data of a single node or used analytical models to study multiple nodes without capturing network behavior.

Understanding the potential energy savings during checkpointing periods as well as the interplay between CPU performance and checkpoint commit speed provides the building blocks for future power-aware checkpointing research. In a broader scope, this work demonstrates that checkpoint events can involve significant amounts of CPU usage depending on the system configuration. We believe this finding could have potential impact on the performance of recently suggested staged checkpoints, in which checkpoints are written locally then copied over the network while the application continues to execute. If one has a system in which the CPU is heavily involved, the copy operation will be slower than expected and will likely interfere significantly with application progress.

Going forward, we will explore energy savings using a fully offloaded InfiniBand network card. We believe the energy savings possible will more closely match those found in the local checkpoint case. We also plan on exploring the use of more traditional parallel file systems in addition to our local storage system presented in this paper. Additionally, we are exploring energy optimizations in other parts of rollback/recovery; for example, the restart and rework phases. Lastly, we plan to explore different checkpoint methods, including uncoordinated checkpointing. Our hypothesis is that uncoordinated checkpointing will benefit from these techniques. However, due to the lack of coordination, the performance implications are not clear.

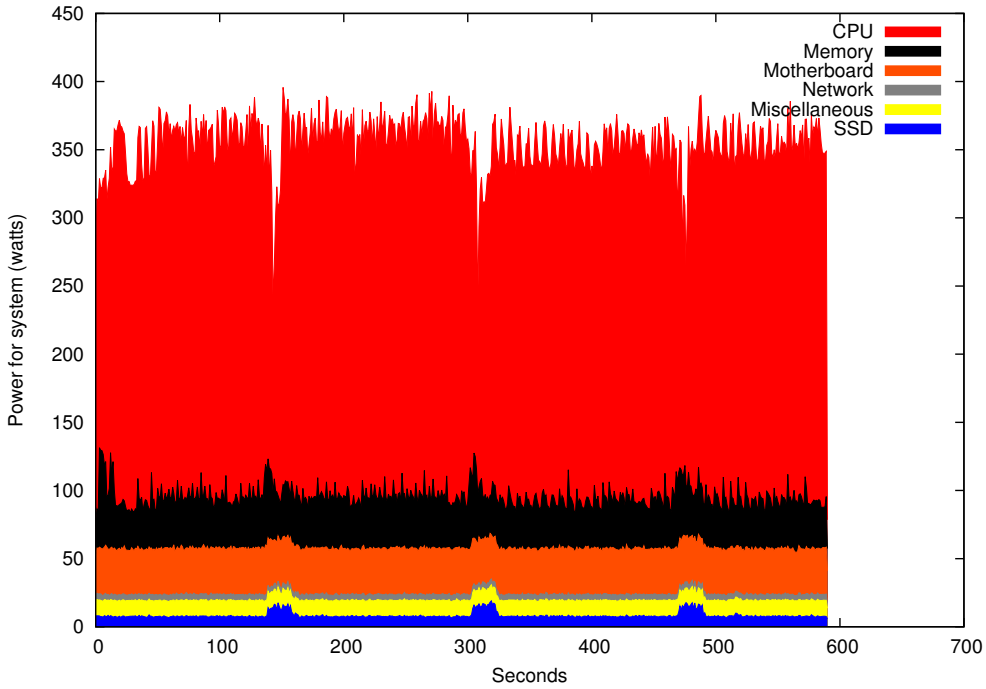


(a) 3.8 Ghz

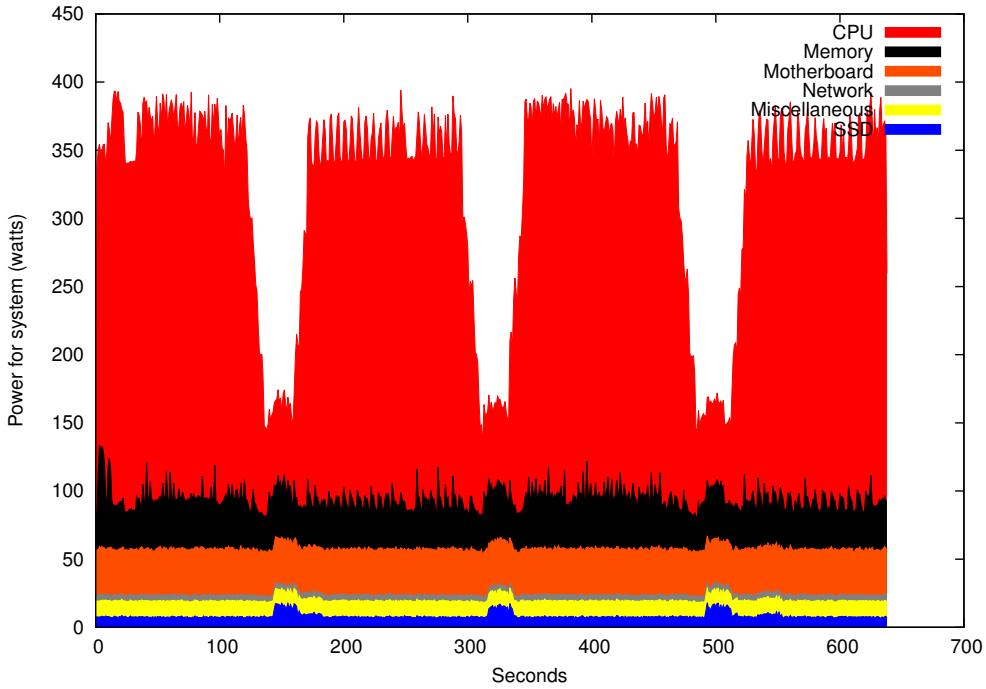


(b) 1.4 Ghz

Figure 37: *Local SSD.* Component level power profile during three coordinated checkpoints of a LAMMPS application run within a 4-node cluster using 16 processes, each process checkpoint is approximately 700MB.

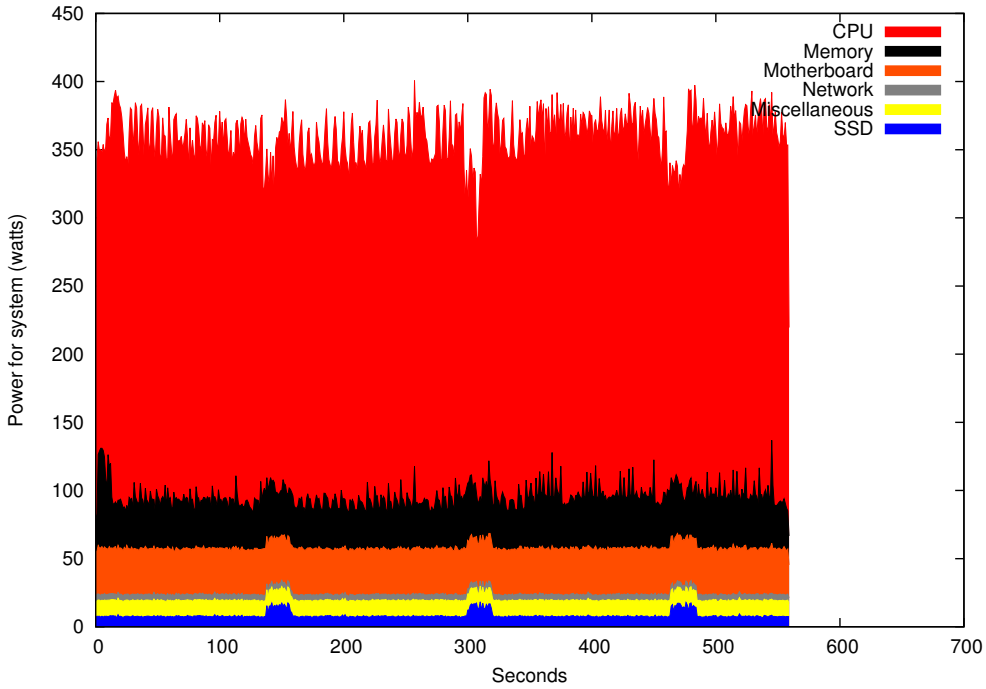


(a) 3.8 Ghz

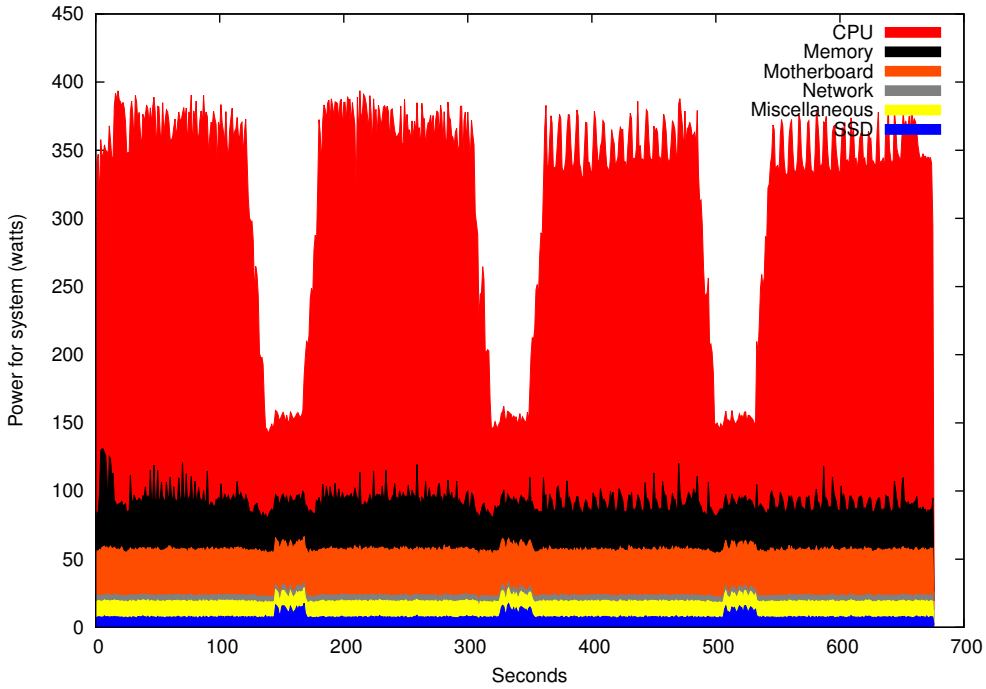


(b) 1.4 Ghz

Figure 38: *Remote SSD using IP over InfiniBand.* Component level power profile during three coordinated checkpoints of a LAMMPS application run within a 4-node cluster using 16 processes, each process checkpoint is approximately 700MB.



(a) 3.8 Ghz



(b) 1.4 Ghz

Figure 39: *Remote SSD using RDMA over InfiniBand.* Component level power profile during three coordinated checkpoints of a LAMMPS application run within a 4-node cluster using 16 processes, each process checkpoint is approximately 700MB.

BIBLIOGRAPHY

[1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM.

[2] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang, K. Joy, Q. Koziol, G. Lofstead, J. S. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wolf, N. Wright, and K. Wu. Scientific discovery at the exascale, a report from the doe ascr 2011 workshop on exascale data management, analysis, and visualization, 2011.

[3] F. M. Al-Athari. Estimation of the mean of truncated exponential distribution. *Journal of Mathematics and Statistics*, 2008.

[4] L. Alvisi, E. Elnozahy, S. Rao, S. Husain, and A. de Mel. An analysis of communication induced checkpointing. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 242 –249, 1999.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.

[6] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 313–322, Dec 2006.

[7] J. F. Bartlett. A nonstop kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 22–29, New York, NY, USA, 1981. ACM.

[8] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, and P. Kogge. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[9] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, Nov. 1987.

[10] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Using group replication for resilience on exascale systems. Rapport de recherche RR-7876, INRIA, Feb. 2012.

[11] D. Briatico, A. Ciuffoletti, and L. Simoncini. A Distributed Domino-Effect Free Recovery Algorithm. In *4th IEEE Symposyum on Reliability in Distributed Software and Database Systems*, 1984.

[12] F. Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA*, 23(3):212–226, 2009.

[13] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Combining Process Replication and Checkpointing for Resilience on Exascale Systems. Rapport de recherche RR-7951, INRIA, May 2012.

[14] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[15] V. Chandra and R. Aitken. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS. In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS '08. IEEE International Symposium on*, pages 114–122, Oct 2008.

[16] K. Chandy and C. Ramamoorthy. Rollback and recovery strategies for computer programs. *Computers, IEEE Transactions on*, C-21(6):546–556, June 1972.

[17] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[18] H. chang Nam, J. Kim, S. Lee, and S. Lee. Probabilistic checkpointing. In *In Proceedings of Intl. Symposium on Fault-Tolerant Computing*, pages 153–160, 1997.

[19] F. Chen, D. A. Koufaty, and X. Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 22–32, New York, NY, USA, 2011. ACM.

[20] J. Chu and V. Kashyap. Transmission of ip over infiniband (ipoib). Technical report, RFC 4391, April, 2006.

[21] X. Cui, B. Mills, T. Znati, and R. Melhem. Shadows on the cloud: An energy-aware, profit maximizing resilience framework for cloud computing. In *International Conference on Cloud Computing and Services Science, CLOSER 2014*, Apr 2014.

[22] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[23] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, Feb. 2006.

[24] M. Diouri, O. Gluck, L. Lefèvre, and F. Cappello. Energy considerations in checkpointing and fault tolerance protocols. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[25] J. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *Proceedings of the 19th International Symposium on Shock Waves*, 1993.

[26] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for HPC. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 615–626, Washington, DC, USA, 2012. IEEE Computer Society.

[27] E. Elnozahy, R. Melhem, and D. Mosse. Energy-efficient duplex and tmr real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 256–266, 2002.

[28] E. Elnozahy and J. Plank. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *Dependable and Secure Computing, IEEE Transactions on*, 1(2):97 – 108, april-june 2004.

[29] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE TRANSACTIONS ON COMPUTERS*, 41:526–531, 1992.

[30] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[31] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[32] C. Engelmann and S. Böhm. Redundant execution of HPC applications with MR-MPI. In *Proceedings of the 10^{th} IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*, pages 31–38, Innsbruck, Austria, Feb. 15-17, 2011. ACTA Press, Calgary, AB, Canada.

[33] C. Engelmann, S. Scott, C. Leangsuksun, and X. He. Symmetric active/active high availability for high-performance computing system services: Accomplishments and limitations. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 813–818, May 2008.

[34] C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He. Towards high availability for high-performance computing system services: Accomplishments and limitations. In *In Proceedings of High Availability and Performance Workshop*, 2006.

[35] K. Ferreira, R. Riesen, P. Bridges, D. Arnold, J. Stearley, J. H. L. III, R. Oldfield, K. Pedretti, and R. Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *SC*, Nov. 2011.

[36] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, and R. Brightwell. rmpi: increasing fault resiliency in a message-passing environment. Technical Report SAND2011-2488, Sandia National Laboratories, Albuquerque, New Mexico, 2011.

[37] K. Ferreira, J. Stearley, J. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. ACM.

[38] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[39] K. B. Ferreira. *Keeping Checkpoint/Restart Viable for Exascale Systems*. PhD thesis, University of New Mexico, Department of Computer Science, Dec. 2011.

[40] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[41] M. Gamell, I. Rodero, M. Parashar, J. C. Bennett, H. Kolla, J. Chen, P.-T. Bremer, A. G. Landge, A. Gyulassy, P. McCormick, S. Pakin, V. Pascucci, and S. Klasky. Exploring power behaviors and trade-offs of in-situ data analytics. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 77:1–77:12, New York, NY, USA, 2013. ACM.

[42] R. Ge, X. Feng, W.-c. Feng, and K. W. Cameron. CPU miser: A performance-directed, run-time system for power-aware clusters. In *International Conference on Parallel Processing (ICPP)*, pages 18–18. IEEE, 2007.

[43] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5), 2010.

[44] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 989–1000, May 2011.

[45] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, Washington, DC, USA, 2011. IEEE Computer Society.

[46] D. Hakkarinen and Z. Chen. Multilevel diskless checkpointing. *Computers, IEEE Transactions on*, 62(4):772–783, April 2013.

[47] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[48] P. Hazucha and C. Svensson. Impact of CMOS technology scaling on the atmospheric neutron soft error rate. *Nuclear Science, IEEE Transactions on*, 47(6):2586–2594, Dec 2000.

[49] J.-M. Helary, A. Mostefaoui, R. Netzer, and M. Raynal. Preventing useless checkpoints in distributed computations. In *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*, pages 183 –190, oct 1997.

[50] C.-h. Hsu and W.-c. Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 1. IEEE Computer Society, 2005.

[51] S. Huang and W. Feng. Energy-efficient cluster computing via accurate workload characterization. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 68–75. IEEE Computer Society, 2009.

[52] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc., 1994.

[53] R. Jejurikar and R. Gupta. Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 78–81, Aug 2004.

[54] Q. Jiang, Y. Luo, and D. Manivannan. An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems. *J. Parallel Distrib. Comput.*, 68(12):1575–1589, Dec. 2008.

[55] S. Kalaiselvi and V. Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*, 25(5):489–510, 2000.

[56] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 33. IEEE Computer Society, 2005.

[57] P. M. Kogge and D. R. Resnick. Yearly update: Exascale projections for 2013. Technical Report SAND2013-9229, Sandia National Laboratories, Albuquerque, New Mexico, 2013.

[58] P. Koopman and J. DeVale. Comparing the robustness of posix operating systems. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 30–, Washington, DC, USA, 1999. IEEE Computer Society.

[59] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Elevier, Inc., 2007.

[60] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[61] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, Apr. 1984.

[62] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[63] J.-C. Laprie. Dependable computing and fault tolerance : Concepts and terminology. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pages 2–, Jun 1995.

[64] J. Laros, K. Pedretti, S. Kelly, W. Shu, and C. Vaughan. Energy based performance tuning for large scale high performance computing systems. In *20th High Performance Computing Symposium*, Orlando Florida, 2012.

[65] J. Laros, P. Pokorny, and D. DeBonis. Powerinsight - a commodity power measurement capability. In *The Third International Workshop on Power Measurement and Profiling in conjunction with IEEE IGCC 2013*, Arlington Va, 2013.

[66] Lawrence Livermore National Laboratory. ASC sequoia benchmark codes. https://asc.llnl.gov/sequoia/benchmarks/, 2013.

[67] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok. Volpexmpi: An mpi library for execution of parallel applications on volatile nodes. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*, pages 124–133. Springer Berlin Heidelberg, 2009.

[68] A. Lefray, T. Ropars, and A. Schiper. Replication for send-deterministic MPI HPC applications. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, FTXS '13, pages 33–40, New York, NY, USA, 2013. ACM.

[69] D. Li, B. de Supinski, M. Schulz, D. Nikolopoulos, and K. Cameron. Strategies for energy efficient resource management of hybrid programming models. *Parallel and Distributed Systems, IEEE Transactions on*, 24(1):144–157, 2013.

[70] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, Aug. 1994.

[71] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Proceedings of the 23rd IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2010*, pages 1–12, New Orleans, LA, USA, Nov. 13-19, 2010. ACM Press, New York, NY, USA.

[72] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 14–14. IEEE, 2006.

[73] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, Apr. 1962.

[74] E. Meneses, O. Sarood, and L. V. Kale. Assessing energy efficiency of fault tolerance protocols for HPC systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 35–42. IEEE, 2012.

[75] B. Mills, R. E. Grant, K. B. Ferreira, and R. Riesen. Evaluating energy saving for checkpoint/restart. In *First International Workshop on Energy Efficient Supercomputing (E2SC) in conjunction with SC13: The International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2013.

[76] B. Mills, T. Znati, and R. Melhem. Shadow computing: An energy-aware fault tolerant computing model. In *In Proceedings of the International Conference on Computing, Networking and Communications (ICNC)*, 2014.

[77] B. Mills, T. Znati, R. Melhem, R. E. Grant, and K. B. Ferreira. Energy consumption of resilience mechanisms in large scale systems. In *Parallel, Distributed and Network-Based Processing (PDP), 22st Euromicro International Conference*, Feb 2014.

[78] M. Mittal and R. Valentine. Performance throttling to reduce IC power consumption, Feb. 17 1998. US Patent 5,719,800.

[79] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[80] X. Ni, E. Meneses, N. Jain, and L. V. Kalé. Acr: Automatic checkpoint/restart for soft and hard error protection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 7:1–7:12, New York, NY, USA, 2013. ACM.

[81] R. Oldfield, S. Arunagiri, P. Teller, S. Seelam, M. Varela, R. Riesen, and P. Roth. Modeling the impact of checkpoints on next-generation systems. In *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, pages 30 –46, sept. 2007.

[82] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth. Modeling the impact of checkpoints on next-generation systems. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, Sept. 2007.

[83] T. Pering, T. Burd, and R. Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system. In *Power Driven Microarchitecture Workshop, attached to ISCA98*, pages 96–101, 1998.

[84] J. Plank and K. Li. Faster checkpointing with n+1 parity. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 288–297, June 1994.

[85] J. Plank and M. Thomason. The average availability of parallel checkpointing systems and its importance in selecting runtime parameters. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 250 –257, 1999.

[86] J. S. Plank, Y. Kim, and J. J. Dongarra. Algorithm-based diskless checkpointing for fault-tolerant matrix operations. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 351–, Washington, DC, USA, 1995. IEEE Computer Society.

[87] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117, 1995.

[88] M. L. Powell and D. L. Presotto. Publishing: a reliable broadcast communication mechanism. In *Proceedings of the ninth ACM symposium on Operating systems principles*, SOSP '83, pages 100–109, New York, NY, USA, 1983. ACM.

[89] X. Qi, D. Zhu, and H. Aydin. Global reliability-aware power management for multiprocessor real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 183–192, Aug 2010.

[90] X. Qi, D. Zhu, and H. Aydin. Global scheduling based reliability-aware power management for multiprocessor real-time systems. *Real-Time Systems*, 47(2):109–142, 2011.

[91] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[92] R. Riesen, K. Ferreira, J. R. Stearley, R. Oldfield, J. H. L. III, K. T. Pedretti, and R. Brightwell. Redundant computing for exascale systems, December 2010.

[93] Rolf Riesen, Kurt Ferreira, Jon Stearley, Ron Oldfield, James H. Laros III, Kevin Pedretti and Ron Brightwell. *Redundant Computing for Exascale Systems.* Sandia National Laboratories, December 2010. Sandia Report SAND2010-8709.

[94] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: making DVS practical for complex HPC applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 460–469. ACM, 2009.

[95] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 139–152, New York, NY, USA, 2011. ACM.

[96] T. Saito, K. Sato, H. Sato, and S. Matsuoka. Energy-aware I/O optimization for checkpoint and restart on a NAND flash memory system. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, pages 41–48. ACM, 2013.

[97] Sandia National Laboratory. Mantevo project home page. https://software.sandia.gov/mantevo, 2010.

[98] O. Sarood, E. Meneses, and L. V. Kale. A 'cool' way of improving the reliability of HPC machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 58:1–58:12, New York, NY, USA, 2013. ACM.

[99] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 19:1–19:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[100] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[101] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, June 2006.

[102] M. Seager. Operational machines: ASCI white. In *7th Workshop on Distributed Supercomputing, Durango, CO*, 2003.

[103] D. Siewiorek and R. Swarz. *The theory and practice of reliable system design*. Digital Press, 1982.

[104] J. Stearley, K. Ferreira, D. Robinson, J. Laros, K. Pedretti, D. Arnold, P. Bridges, and R. Riesen. Does partial replication pay off? In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, June 2012.

[105] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, Aug. 1985.

[106] A. Tiwari, M. Laurenzano, J. Peraza, L. Carrington, and A. Snavely. Green queue: Customized large-scale clock frequency scaling. In *2012 Second International Conference on Cloud and Green Computing (CGC)*, pages 260–267. IEEE, 2012.

[107] N. Vallina-Rodriguez and J. Crowcroft. Energy management techniques in modern mobile handsets. *Communications Surveys Tutorials, IEEE*, 15(1):179–198, First 2013.

[108] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. *SIGOPS Oper. Syst. Rev.*, 41(6):59–72, Oct. 2007.

[109] Y.-M. Wang, P.-Y. Chung, I.-J. Lin, and W. K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(5):546–554, May 1995.

[110] Y.-M. Wang, Y. Huang, and W. Fuchs. Progressive retry for software error recovery in distributed systems. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 138–144, June 1993.

[111] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka. Integrating dynamic pricing of electricity into energy aware scheduling for HPC systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 60:1–60:11, New York, NY, USA, 2013. ACM.

[112] G. Zheng, L. Shi, and L. Kale. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Cluster Computing, 2004 IEEE International Conference on*, pages 93–103, Sept 2004.

[113] D. Zhu, R. Melhem, and B. R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):686–700, July 2003.

[114] D. Zhu, R. Melhem, D. Mosse, and E. Elnozahy. Analysis of an energy efficient optimistic tmr scheme. In *Parallel and Distributed Systems, 2004. ICPADS 2004. Proceedings. Tenth International Conference on*, pages 559–568. IEEE, 2004.

[115] D. Zhu, R. Melhem, and D. Moss. The effects of energy management on reliability in real-time embedded systems. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 35–40, 2004.